ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Grado en Ingeniería Informática – Tecnologías Informáticas

GOBIERNO DE APIS REST BASADO EN SLAS

Realizado por

ANTONIO GÁMEZ DÍAZ

Dirigido por

PABLO FERNÁNDEZ MONTES

Departamento

LENGUAJES Y SISTEMAS INFORMÁTICOS

Sevilla, 2 de junio de 2015

Publicado en mayo 2015 por
Antonio Gámez Díaz

http://personal.us.es/agamez2
agamez2@us.es

Yo, D. Antonio Gámez Díaz con NIF número 77808176G,

DECLARO

mi autoría del trabajo que se presenta en la memoria de este trabajo fin de grado que tiene por título:

Gobierno de APIs REST basado en SLAs

Lo cual firmo,

Fdo. D. Antonio Gámez Díaz
en la Universidad de Sevilla
27/05/2015

Dedicado mi madre, Rosi. Sin todo su esfuerzo no hubiera llegado nunca hasta donde estoy. Gracias por todo.

Agradecimientos

Quisiera agradecer a varias personas y entidades la ayuda que me han prestado en la realización de este Trabajo de Fin de Grado. Entre ellas, y en primer lugar, a mi tutor, Pablo Fernández, por todas las horas de reuniones y revisiones que hemos tenido y, especialmente, por confiar en mí desde el primer momento: un trabajo distinto a lo que se suele ofertar, un apoyo continuo y muchas ganas de que participe en la línea del grupo ISA. En este sentido he de agradecer también el interés de Antonio Ruiz, que ha hecho posible que elabore una publicación para un congreso nacional exponiendo los resultados de este trabajo y ha mostrado un continuo interés en mi desarrollo dentro del grupo. Gracias a ellos he comenzado a introducirme en el mundo de la investigación, con gran ilusión, ganas de progresar y esperando que el contrato como *Técnico Especialista de Apoyo a la Investigación* sólo sea el primer paso de una larga carrera.

Asimismo quiero aprovechar esta sección para mostrar mi enorme agradecimiento a mis padres, y en especial, a mi madre, por la infinita paciencia que ha tenido y por ser siempre un apoyo y referente en todos los momentos. Gracias a ella conocí el mundo de la informática cuando era pequeño y ha conseguido que llegue a ser lo que siempre he deseado: Ingeniero Informático. ¡Muchas gracias por todo!

Resumen

Contexto

La evolución de la industria hacia un modelo de software como servicio ha favorecido la aparición de un mercado de APIs en continuo crecimiento. En este contexto es necesario para los desarrolladores de APIs dar soporte a planes de precio y gestión de niveles de servicio. Para desacoplar la lógica de la API de estas tareas se han creado plataformas de gestión denominadas *API Gateways*. Sin embargo, estas plataformas presentan bastantes limitaciones debido a que no establecen un modelo de SLA explícito.

Objetivos

En este trabajo se pretende realizar un estudio del estado del arte de las APIs y SaaS existentes en el mercado para comprobar las necesidades en cuanto a planes de precio y gestión de peticiones, además de estudiar los *API Gateways* para analizar sus funcionalidades y así construir una herramienta que proporcione soporte a las APIs para cubrir tales necesidades.

Resultados

La investigación realizada durante este trabajo ha dado como resultado una *framework* que cualquier API puede incorporar, personalizable con plantillas de SLA, que dota a la API de forma automática de un sistema de gestión de autorización de peticiones basadas en cuotas establecidas en los acuerdos. Además se ha realizado un estudio de las necesidades de las APIs y características de algunos *API Gateways*.

Conclusiones

En un contexto donde existe un mercado abierto y en crecimiento de APIs con modelos de precio muy variados, el *framework* desarrollado establece los fundamentos de un gobierno automatizado de las APIs, simplificando y abriendo la puerta a otros que tengan en cuenta elementos como los costes de infraestructura y modelos avanzados de penalizaciones. Las contribuciones de este trabajo han permitido la elaboración de un artículo para un congreso nacional y han servido de base para complementar material de algunas prácticas docentes en asignaturas de la Escuela.

Índice general

Índice de figuras

Índice de cuadros

Códigos

Parte I

Introducción

Introducción

A long time ago in a galaxy far, far away...

Star Wars (1977),
Film

En este capítulo se pretende dar comienzo al presente trabajo de fin de grado, introduciendo la estructura del documento y presentando el proyecto a través de la contextualización del mismo en el paradigma de la Computación Orientada a Servicios y en las líneas investigadoras existentes.

1.1 Contexto

1.1.1 El mundo del Software como Servicio

El término *microservicio* ha ido adquiriendo notoriedad en un contexto de la Ingeniería del Software donde el paradigma de la Computación Orientada a Servicios cuenta mayor peso actualmente. Se observa que desde una década atrás se ha hablado de *arquitectura orientada a servicios (SOA)* y cómo sus principios aportan valor a la organización; dichos pilares de *SOA* incluyen el bajo acoplamiento, abstracción, reusabilidad, autonomía e interoperatibilidad. En definitiva, elementos imprescindibles para alejarse de las arquitecturas monolíticas convencionales.

Bajo la denominación de microservicio se incluyen desde servicios web de tipo *REST* o *SOAP* hasta nuevos modelos de distribución de software bajo demanda, conocidos como *SaaS*. De ahí que la proliferación de estos nuevos modelos han llegado a convertirlos en estándar *de facto* en el paradigma de la producción de software. En este contexto, las posibilidades de personalización y adaptación del servicio a cada cliente van más allá de la necesidad de integración y orquestación del servicio.

Históricamente en entornos empresariales se han extendido numerosas herramientas que ofrecen soporte a los principales pilares de una arquitectura orientada a servicios; por ejemplo: integración, orquestación, enrutado y monitorización. Casi todas ellas están bajo el término *Enterprise Service Bus (ESB)* o *Enterprise Application Integration (EAI)*. Actualmente las necesidades de la organización se mueven desde una integración centralizada hacia un contexto distribuido y flexible. Es por ello que ha existido un cambio de visión de la arquitectura orientada a servicios; los microservicios deben seguir dando soporte a ciertas necesidades, como la integración, publicación y descubrimiento de servicios, administración y despliegue escalable.

Casi todas estas necesidades están siendo ya cubiertas en mayor o menor medida, sin embargo, la gestión y automatización de contratos de servicios está aún sin resolver de una forma consensuada; entendiendo contrato como el compromiso y acuerdo alcanzados entre el consumidor y el productor del servicio. Para esta tarea cada organización emplea sus propias estrategias, pero casi todas confluyen en sistemas de monitorización ligados a un acuerdo en lenguaje natural.

En este contexto, el presente trabajo se enfoca en ofrecer una primera aproximación tecnológica a la aplicación de *acuerdos a nivel de servicio* en APIs REST.

1.1.2 Dónde se sitúa el proyecto

Este proyecto de fin de grado se enmarca dentro de las líneas investigadoras del grupo de *Ingeniería del Software Aplicada* (TIC205) de la Universidad de Sevilla, coordinado por el Dr. Antonio Ruiz Cortés; en un contexto de Líneas de Productos, Métodos Formales Aplicados a Familias de Procesos y de Productos, Ingeniería de Requisitos, Sistemas Complejos y, sobre todo, Computación Orientada a Servicios.

En concreto, el proyecto guarda estrecha relación con el proyecto financiado por el Ministerio de Economía y Competitividad *Tecnologías Avanzadas para Procesos como Servicios* (TIN2012-32273). Este proyecto incluye entre sus principales metas el soporte a servicios que puedan personalizarse de acuerdo a las necesidades del cliente y correr en la nube, un claro caso de necesidad de aplicación de *acuerdos a nivel de servicio*.

El grupo de investigación lleva trabajando desde hace varios años en la formalización de acuerdos. En este marco, destacan artículos como [17], automatizando el análisis de conflictos de *SLA* tratándolo como *problemas de satisfacción de restricciones (CSP)*; [14], ofreciendo una explicación de las violaciones al acuerdo en tiempo de ejecución mediante una herramienta de monitorización; [15], dando un modelo para la creación de *SLA* con penalizaciones o recompensas según el cumplimiento de *objetivos de nivel de servicio (SLO)* y [4], dando herramientas para la edición y validación de documentos de acuerdos formalizados en lenguajes estándares como *WS-Agreement*.

Varias tesis doctorales han sido realizadas también en este campo, por ejemplo, [8], donde se estudia la relación entre servicios web y la programación con restricciones; y [13], donde se define un lenguaje alternativo para la formalización de acuerdos y se realiza la correspondencia con operaciones de análisis de tipo *CSP*.

1.2 Objetivos

1.2.1 Objetivos técnicos

Las APIs son cada vez más populares para la construcción aplicaciones basadas en servicios (SBA), asimismo, existe cierta tendencia creciente en el uso de *API Gatways* para facilitar la gestión de funciones de la API. Estos ofrecen funcionalidad de gestión de API como el soporte de planes de precios, autenticación de usuarios, control de versiones y el almacenamiento en caché de las respuestas. Mucha de la información

que un *API Gateway* necesita ya se encuentra disponible en acuerdos a nivel de servicio (SLA) que los proveedores utilizan para describir los derechos y las obligaciones de las partes involucradas en el servicio. Desafortunadamente, los *API Gateways* actuales no utilizan ningún modelo de representación de SLA explícito ni hace uso de la tecnología existente para su procesamiento, desaprovechando así posibles oportunidades.

Existen numerosas razones por las que es importante disponer un SLA explícito; en primer lugar, nos permite desacoplar la lógica del negocio (e.g. modelos de cobro, compensaciones) con la lógica tecnológica de la API. Esto ya supone una enorme ventaja competitiva, pues los costes de operación y mantenimiento de estos sistemas serán notablemente menores. Por otra parte, un SLA explícito puede evolucionar en el tiempo, pero si se dispone de una biblioteca estándar para el procesado y razonamiento con acuerdos, la aplicación no necesitará cambios profundos para soportar tales cambios. En definitiva, el uso de acuerdos explícitos nos acerca a lógicas desacopladas, reducción de costes y aumento de flexibilidad en la evolución de la API.

En relación a todo lo anterior podemos identificar una serie de objetivos ue servirán de punto de referencia durante todo este proyecto.

Objetivo 1. Analizar el *estado del arte* de APIs y SaaS existentes en el mercado.

Objetivo 2. Analizar la situación de los *API Gateways* actuales para ver si existe alguna alternativa existente a nuestra propuesta.

Objetivo 3. Desarrollar una solución tecnológica inicial, susceptible de extensiones futuras, para dar soporte a APIs diseñadas en Java en las tareas de cobro y gestión de peticiones por usuario a través de la existencia de acuerdos implícitos.

Objetivo 4. Fomentar el uso de acuerdos explícitos ofreciendo una solución fácilmente aplicable a una API existente.

1.2.2 Objetivos académicos

La elaboración del presente trabajo de fin de grado tiene ha servido para fomentar el aprendizaje y desarrollo personal y profesional del autor, por ello, se han establecido objetivos más a allá de los meramente técnicos.

- **Objetivo 1.** Colaborar con un grupo de investigación para así poder observar otra faceta de la Universidad, desconocida para muchos.

- **Objetivo 2.** Sentar las bases de una futura línea de trabajo: la gestión del precio en APIs a través de acuerdos explícitos.
- **Objetivo 3.** Aprendizaje profesional y personal al trabajar con un equipo multidisciplinar de técnicos e investigadores para confluir en un objetivo común.

1.3 Estructura del documento

El presente documento está dividido en siete bloques, tres documentos anexos y bibliografía utilizada. A continuación se detalla el contenido de cada uno de estos bloques:

- **Introducción**: en este primer bloque se contextualiza el proyecto y se presentan los principales objetivos identificados.
- **Metodología**: aquí se expone cómo se ha enfocado este proyecto, en cuanto a metodología empleada y planificación por hitos. Finalmente se realiza un estudio de los costes incurridos durante todas las fases.
- **Análisis**: durante este bloque se efectúa un análisis del estado del arte en el paradigma del software como servicio y se realiza un estudio tecnológico incial sobre los *frameworks* Java existentes en el mercado.
- **Diseño**: aquí se presenta la arquitectura de la solución propuesta en este trabajo y se introduce un caso de estudio que servirá de ejemplo de los resultados obtenidos.
- **Implementación**: en este bloque se comenta el desarrollo realizado para satisfacer los requisitos identificados previamente. Posteriormente se realiza una validación y comparación de las diversas soluciones.
- **Manuales**: este bloque presenta una guía para que cualquier desarrollador de APIs pudiera integrar la solución que presentamos en este trabajo, partiendo para ello, del caso de estudio que se presentó previamente.
- **Conclusiones**: en este último bloque se realiza un análisis y valoración final de los resultados obtenidos obtenidos al completar el proyecto y se estudian las posibles extensiones o trabajo futuro.

Parte II

Metodología

Planificación

Adventure is just bad planning.

Roald Amundsen (1872-1928),
Norwegian explorer

La planificación para un proyecto es el estudio del tiempo, necesidades y distribución de tareas que debemos tener para la realización de las distintas fases o etapas de un proyecto. La medición final se basa en la contabilización de las horas que finalmente nos ha llevado realizar cada una de las tareas y subtareas.

Este trabajo de fin de grado se enmarca dentro de una línea de investigación en curso, por lo que presenta unas peculiaridades derivadas de las incertidumbres propias de tales actividades.

2.1 METODOLOGÍA DE DESARROLLO

Se ha usado una metodología ligera, incremental e iterativa adaptada a las peculiaridades del proyecto donde se ha optado por establecer un calendario de reuniones breves semanales, en las que se realice el seguimiento de las tareas a realizar identificadas. En caso de no haber cubierto los objetivos para una reunión determinada, se analizan las causas y se pospone a la siguiente semana.

El control de tareas ha sido realizado con la aplicación web de Assembla, una aplicación realizada sobre *Ruby on Rails* (con algunas partes en *Python*) que ofrece espacios de trabajo para los desarrolladores software, con soporte SVN, GIT, Mercurial, blog, wiki, gestión de bugs y tickets, gestión de usuarios, múltiples proyectos, repositorio para documentación, anotación de imágenes, chat, alertas, etc.

Para las comunicaciones y compartición de archivos entre se ha empleado, fundamentalmente, el *stack* integrado por *Google Drive*, *Google Docs* y *Microsoft OneNote*.

2.2 RESUMEN TEMPORAL DEL PROYECTO

En la figura §2.1 se aprecia resumen de la temporalización de este proyecto.

Resumen del proyecto	
Fecha de inicio	21/10/2014
Fecha de fin	2/06/2015
Periodicidad de las revisiones	1 semanas
Carga de trabajo semanal	12 horas
Horas totales previstas	320 horas
Horas finales	343,8 horas

Cuadro 2.1: Tabla resumen de tiempos y planificación

2.3 INFORME DE TIEMPOS DEL PROYECTO

De las iteraciones semanales anteriormente mencionadas, extraemos los hitos fundamentales del proyecto, tal y como se aprecia en §2.2

Resumen de iteraciones	
Iteración 1: toma de contacto inicial	21/10/14 a 18/11/14
Iteración 2: puesta en marcha tecnológica	18/11/14 a 2/12/14
Iteración 3: formación en Ingeniería del Software	2/12/14 a 23/12/14
Iteración 4: estudio de soluciones existentes	23/12/14 a 19/02/15
Iteración 5: diseño	19/02/15 a 9/04/15
Iteración 6: elaboración documentos derivados	9/04//15 a 8/05/15
Iteración 7: implementación y documentación	8/05/15 a 18/05/15

Cuadro 2.2: Planificación temporal de iteraciones

En concreto, cada iteración de las mostradas en la figura §2.2 se detalla a continuación:

- **Iteración 1**: investigación y documentación del *background* investigador a modo de toma de contacto.
- **Iteración 2**: instalación de stack tecnológico y creación de pruebas de concepto para un estudio tecnológico.
- **Iteración 3**: documentación y formación en conceptos de la Ingeniería del Software (patrones de diseño y documentación de autores clásicos).
- **Iteración 4**: estudio detallado de herramientas del grupo y propuesta de mejora de las soluciones existentes para la gestión de SLA.
- **Iteración 5**: uso de biblioteca desarrollada por el grupo y aplicación a casos concretos.
- **Iteración 6**: elaboración de memoria para beca de iniciación a la investigación y artículo para congreso nacional.
- **Iteración 7**: implementación de la solución propuesta y finalización de la documentación.

Para mejorar la visión y perspectiva global del desarrollo del proyecto, se han volcado las tareas realizadas, agrupadas por hitos y se han representado en un diagrama de Gantt, tal y como aparece en la Figura §2.1.

Se puede observar una desviación en unas 20 horas respecto de la planificación inicial, pero es justificable debido a especial carácter de este proyecto. Se había estimado en algo más de 300 horas al valorar la complejidad de enfrentarse por primera vez a tareas investigadoras siguiendo la línea de un grupo de investigación, pero aun así, no se ha ajustado a lo planificado. El hecho de haber desarrollado un artículo para un congreso nacional y centrar algunos esfuerzos en mostrar un prototipo funcional del uso de la biblioteca desarrollada por el grupo por necesidades docentes hayan conducido a ese incremento de la horas dedicadas.

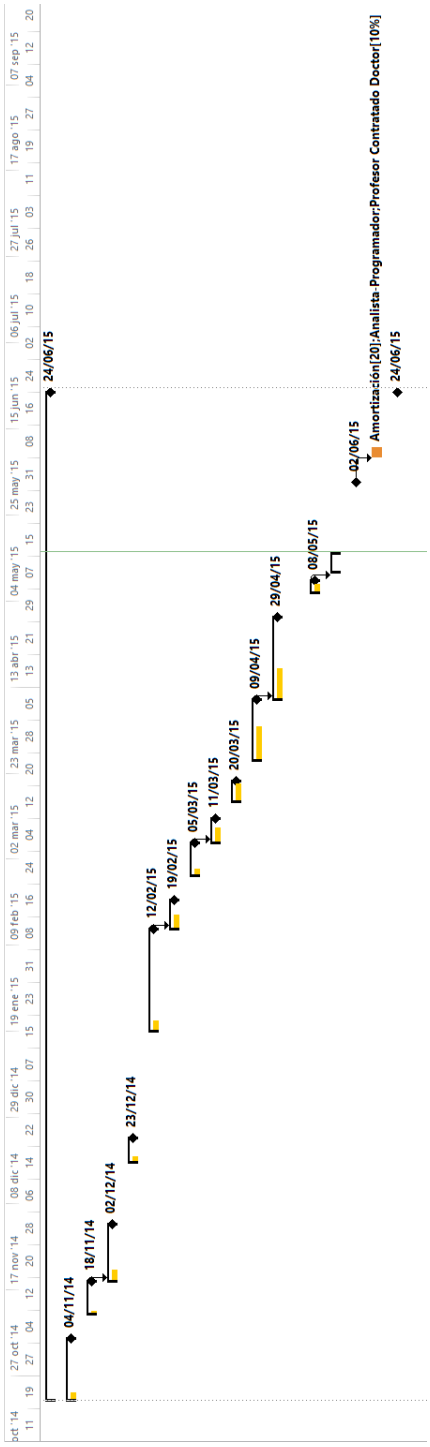

Figura 2.1: Diagrama de Gantt de hitos.

Estimación de costes

Goals are dreams with deadlines

Diana Scharf Hunt (1990),
Author

En este capítulo se calculan los costes estimados en base a la planificación realizada, teniendo conocimiento de los salarios de cada rol involucrado en el proyecto, las horas planificadas por cada uno y el coste de las licencias utilizadas en caso de haberlas.

3.1 Costes de personal

Al tratarse de un proyecto con una fuerte componente investigadora asumiremos que para la realización del mismo han sido necesarios tres roles diferentes: un *Profesor Contratado Doctor* para la coordinación del proyecto, un *Ayudante de Investigación* para las tareas técnicas derivadas directamente con la investigación y un *Analista-Programador* para la ejecución de la solución propuesta.

El cálculo de estos costes se basa en dos convenios:

- *XVI CONVENIO COLECTIVO ESTATAL DE EMPRESAS DE CONSULTORÍA Y ESTUDIOS DE MERCADOS Y DE LA OPINIÓN PÚBLICA* [2], que regula el salario mínimo para un *Analista-Programador*.
- *XIII CONVENIO COLECTIVO DE ÁMBITO ESTATAL PARA LOS CENTROS DE EDUCACIÓN UNIVERSITARIA E INVESTIGACIÓN* [3], que regula el salario mínimo de los puestos del *Personal Docente e Investigador*.

Se determina para cada uno de los puestos el salario en €/h:

- **Profesor Contratado Doctor**: 1.573,93 €/ 160 h = 9,84 €/h
- **Analista-Programador**: 1.468,54 €/ 160 h = 9,18 €/h

A estos costes se les debe añadir los costes de la Seguridad Social a cargo de la empresa. Estos se calculan aplicando unos porcentajes que se determinan según el tipo de contrato y puesto de trabajo, aplicándose sobre la base de cotización de cada trabajador, que se calcula como la suma de todos los conceptos salariales de la nómina del trabajador, que se corresponde con las cifras obtenidas en el párrafo anterior. La seguridad social a cargo de la empresa cubre los siguientes riesgos:

- **Contingencias comunes**: contempla el riesgo de enfermedad común del trabajador, por motivos ajenos al trabajo. Este coste se calcula aplicando un 23,6% a la base de cotización de cada trabajador.
- **Desempleo**: cubre el riesgo de que el trabajador se quede en paro, ya sea por despido o por fin del contrato. Para los contratos temporales a tiempo parcial (, como es el caso que nos ocupa, se calcula aplicando un 7,7% sobre la base de cotización.

- **FOGASA**: son las siglas del *Fondo de Garantía Salarial*. Este coste lo asume íntegramente la empresa y cubre el riesgo de insolvencia de las empresas. Se calcula aplicando un 0,2 %sobre la base de cotización.
- **Riesgo profesional**: cubre el riesgo de accidentes laborales y enfermedades profesionales delos trabajadores. Es un coste que asume íntegramente la empresa. Se calcula aplicando un porcentaje que varía, dependiendo de la actividad económica que desarrolle la empresa. Para ello se toma como referencia la tabla del CNAE publicada en el BOE del 24 de Diciembre del 2009. En ésta tabla se puede contemplar que para la actividad Programación, consultoría y otras actividades relacionadas con la informática (código de CNAE: 62) le corresponde un tipo del 1,65 % sobre la base de cotización. Este porcentaje se descompone en los siguientes riesgos:
  - **IT**: contempla el riesgo de incapacidad temporal derivado del trabajo (0,65 %).
  - **IIMS**: es el porcentaje que cubre el riesgo de incapacidad permanente, muerte y supervivencia derivados del trabajo (1 %).
- **Formación profesional**: se trata de una aportación de la empresa para los costes de formación permanente de los empleados. Se calcula aplicando un 0,6 % sobre la base de cotización.

Por tanto la seguridad social a cargo de la empresa supone en total el siguiente porcentaje sobre la base de cotización de cada trabajador:

$$23{,}6\% + 7{,}7\% + 0{,}2\% + 1{,}65\% + 0{,}6\% = 33{,}75\%$$

Los costes salariales quedarían como sigue:

- **Profesor Contratado Doctor**: 9,84 €/h * 1,3375 = 13,15 €/h
- **Analista-Programador**: 9,18 €/h * 1.3375 = 12,28 €/h

En base a la dedicación de estos recursos a las tareas del proyecto, los costes de personal asociados ascienden a:

- **Profesor Contratado Doctor**: 13,15 €/h * 31,75 = 417,74
- **Analista-Programador**: 12,28 €/h * 312,03 = 3.831,73

3.2 Costes materiales

3.2.1 Hardware

Durante el desarrollo será necesario el uso de un equipo informático portátil de 1382€, cuyo uso se repartirá entre los integrantes del equipo teniendo en cuenta la amortización.

Cualquier activo fijo, bien material o inmaterial, que pasa a formar parte de la estructura de actividad de una sociedad sufre como consecuencia del paso del tiempo una depreciación o pérdida de valor. Cosa que supone el hecho contable y fiscal de la amortización. Estas amortizaciones aminoran la base imponible del *Impuesto de Sociedades* o *Estimación Directa del Impuesto sobre la Renta de las Personas Físicas*. En este sentido y puesto que es en el momento en el que se incorpora el activo al patrimonio de la empresa cuando se debe decidir el sistema de amortización que se va a utilizar, y que dichos sistemas proporcionan cuotas de amortización, y por tanto ahorros fiscales, se ha de tener un especial cuidado en su elección.

Para calcular la cantidad que se le asigna a cada año se pueden utilizar distintos métodos, de los cuales se ha seleccionado el método de amortización lineal o de cuotas fijas, que reparte de forma equitativa la amortización del activo a lo largo de los años de su vida útil. Los conceptos que maneja este método son:

- **Valor residual**: valor neto que de ellos se obtendría vendiéndolos en el mercado vigente a la fecha de valuación cuando ha finalizado su vida útil, operativa o tecnológica.
- **Valor de reemplazo**: valor de compra del bien equivalente a la fecha del relevamiento.
- **Periodo de vida útil**: tiempo en años que el bien puede ser utilizado normalmente, con mantenimiento adecuado, en buenas condiciones operativas y tecnológicas.

En base a estas definiciones, se establecen tales parámetros:

- **Valor residual**: 200 €.

- **Valor de reemplazo**: 1382 €- 200 €= 1182 €.
- **Periodo de vida útil**: 4 años (8400 h).

Amortización: 1182 € / 8400 h = 0,14 €/h.

Considerando que el equipo ha sido empleado durante 303,79 horas para realizar la casi totalidad de las tareas que integran el proyecto, el coste total de la infraestructura informática asciende a:

0,14 €/h * 303,79 h = 42,53 €

3.2.2 Software

Para el desarrollo del presente proyecto ha sido necesario el uso de las siguientes herramientas software:

- **Assembla**: para proyectos de desarrollo e investigación sin ánimo de lucro, coste 0 €.
- **Google Drive**: capa gratuita sin características empresariales, coste 0 €.
- **IEEE Xplore**: licencia universitaria adquirida por la Universidad de Sevilla, coste 0 €.
- **IntelliJ IDEA 14.1.1**: licencia universitaria al tratarse de un proyecto de investigación, coste 0 €.
- **Mendeley**: licencia universitaria adquirida por la Universidad de Sevilla, coste 0 €.
- **TeXnicCenter**: gratuito, coste 0 €.

Por tanto, el coste directo correspondiente a software es 0 €.

3.3 Costes indirectos

En base al conocimiento adquirido a partir de la experiencia durante el desarrollo de otros proyectos, decidimos establecer los costes indirectos como el 12,5% de los costes directos. Habiendo valorado estos últimos en 4292 €, tenemos:

$$4292 * 0.125 = 377{,}99\ €$$

3.4 Resumen de costes del proyecto

Resumen del proyecto	
Costes directos	4292 €
Costes de personal	4.249,47 €
Costes materiales	42,53 €
Costes indirectos	377,99 €
TOTAL	4669,99 €

Cuadro 3.1: Tabla resumen de costes

Parte III

Análisis

Estado del arte

Study nature, love nature, stay close to nature. It will never fail you.

Frank Lloyd Wright (1867–1959),
American architect

Comenzaremos analizando algunas de las APIs y soluciones de Software como Servicio más conocidas y analizaremos sus características. Posteriormente realizaremos un recorrido por las numerosas soluciones de gateway para APIs y veremos una comparativa haciendo especial hincapié en las funciones esenciales que se esperan. Más tarde detallaremos la puesta en marcha de una API existente bajo Mashape, intentando analizar la funcionalidad que nos ofrece.

4.1 Introducción

Es un hecho que los nuevos modelos de distribución de software y computación han ido ganando terreno en el campo empresarial; de hecho, es muy frecuente ver blogs no técnicos hablando de *Software como Servicio (SaaS)* y APIs. Por eso, es importante conocer e intentar definir estos términos.
Saas puede entenderse como aquel modelo de distribución de software donde una empresa ofrece el mantenimiento, soporte y operación del software para su utilización por varios clientes durante el tiempo que hayan contratado el servicio. El usuario paga por el uso, por la infraestructura necesaria (CPD, servidores, sistemas de almacenamiento, sistemas de seguridad, etc.) para el correcto funcionamiento de la aplicación y por el mantenimiento (nuevas versiones, corrección de bugs, almacenamiento necesario, etc.) de la infraestructura y aplicación.

Por otro lado las APIs, si bien son conceptos completamente distintos, guardan mucha relación con lo anterior. Un cliente cuando adquiere una solución *SaaS* está llevando sus datos a las instalaciones de un tercero y quizás este sea el principal temor y causa de la resistencia al cambio de muchos empresarios. Y aquí entran en juego las APIs, pues se ofrece al cliente la posibilidad de acceder a sus datos de una manera programática y flexible.
No obstante, el enfoque de API que estudiaremos aquí no es el de aportar valor añadido a una solución *SaaS*, sino el de aquella API que tiene valor *per se* y por ello es susceptible de tener un modelo de cobro.

Todo este estudio preliminar guarda estrecha relación con lo que se presentará en esta memoria, pues conocer el paradigma actual es esencial para poder proponer nuevas soluciones basadas en la investigación sobre la formalización y razonamiento con Acuerdos a Nivel de Servicio.

En concreto, dedicaremos este capítulo a estudiar varias APIs y soluciones *SaaS*, dando una descripción general y detallando los planes que ofrecen. Es una tarea difícil el seleccionar un número reducido de servicios para analizarlos aquí y existen multitud de criterios para la selección; aquí presentamos una pequeña selección arbitraria, centrándonos en ilustrar la diversidad existente. No obstante, podría ser ampliada tanto como quisiéramos, pues cada día se desarrollan nuevas APIs y salen a luz nuevos *SaaS*.

4.2 Estudio de APIs

4.2.1 AlchemyAPI

- Es una empresa perteneciente a IBM que ofrece servicios de procesamiento de lenguaje natural (*aka NLP*) y reconocimiento de imágenes. En el primer campo, dado un texto, permite extraer y clasificar entidades, analizar sentimientos, extraer relaciones, taxonomías, reconocer idiomas, etc. En el segundo, dada una imagen, permite clasificarla y etiquetarla y, si además contiene caras, devuelve las coordenadas junto al género, rango de edad y, si se trata de un rostro conocido, lo indica y devuelve metadatos de la persona reconocida.

- Existen 3 planes basados exclusivamente en el volumen de peticiones diarias realizadas, oscilando entre las 90.000 y 3 millones.

- Hay una capa gratuita de 1000 peticiones diarias y ofrece la posibilidad de contactar para planes personalizados con alto volumen de demanda.

- Además de lo anterior, se puede seleccionar de forma independiente el paquete de rendimiento y soporte deseado: desde la capa gratuita con soporte vía email y una conexión concurrente, podemos llegar a soporte telefónico, *uptime* garantizado por SLA y hasta 50 conexiones concurrentes.

- Finalmente, para un uso esporádico, se establece un modelo de cobro alternativo *pay-as-you-go*, segmentado en rangos de 250k, 750k, 1M, 3M y 5M de transacciones.

4.2.2 Algorithmia

- Esta compañía pretende construir un mercado donde los desarrolladores construyan y consuman algoritmos. Podemos encontrar desde implementaciones de *Dijkstra* para caminos mínimos hasta técnicas estadísticas para *clustering* en el procesamiento de lenguaje natural.

- El modelo de cobro es algo innovador; se pretende que el usuario que desarrolla un algoritmo ponga precio a la llamada del mismo y luego se tarifique por el tiempo de cómputo.

- Para ello, se establece un modelo de créditos (un dólar equivale a unos 10k créditos) que se compran a la compañía y el usuario los consume como desee.
- No tiene capa gratuita, aunque sí se bonifica a los nuevos usuarios con 10k créditos iniciales. Existen planes personalizados para grandes cantidades.

4.2.3 FlightStats

- Esta organización ofrece información completa sobre los vuelos y aeropuertos, de forma que podemos obtener planes de vuelos, planificaciones, retrasos, aterrizajes, vuelos cerca de una zona, etc.
- El modelo de cobro es sencillo: cada recurso tiene un coste por transacción. De esta forma, por ejemplo, conseguir los vuelos cercanos dado un punto es más barato que listar todos los vuelos planificados para un aeropuerto.
- Por otro lado, se ofrecen recursos *premium* (como alertas de vuelo o información de las pantallas *FIDS*[1]) de forma personalizada hablando con un asistente de ventas.
- No existe una capa gratuita, aunque sí nos ofrecen 20k peticiones repartidas por tipo para que podamos probar antes de lanzar nuestra aplicación.

4.2.4 Groupdocs

- Esta compañía centra sus servicios en la gestión de documentos colaborativos con tres modelos de negocio: vender sus bibliotecas en Java/.NET, usar su solución *SaaS* y ofrecer sus servicios como API. Nos centraremos en este último modelo.
- Se ofrecen 7 productos: visor de documentos, creador de anotaciones y marcas para documentos, conversor a distintos formatos, ensamblador de documentos partiendo de plantillas, firma digital de documentos, comparación de documentos.
- Cada uno de estos productos tiene una API asociada y el cobro es independiente. Cada API tiene varios planes asociados en función del volumen de operaciones realizadas.

[1] Flight information display system.

- Por cada API, el concepto de operación es distinto; por ejemplo, para las visualizaciones sólo cuenta cada documento visto, sin importar el número de visualizaciones o usuarios. Sin embargo, para una comparación, se considera operación a cada llamada que realice tal comparación.
- Existen 6 planes por cada API, donde el número de operaciones oscila entre las 750 y las 100k mensuales.
- Se ofrece una capa gratuita de prueba con 100 operaciones mensuales y otras limitaciones, además de planes personalizados para clientes con mayor volumen de operaciones.

4.2.5 Kraken

- Esta empresa se dedica en exclusiva a la compresión de imágenes manteniendo la calidad aparente para reducir los costes derivados del almacenamiento y ancho de banda.
- Existen 5 planes basados en la cantidad de bytes subidos para ser comprimidos. De esta forma, podemos llegar a comprimir desde 500MB hasta 60GB de imágenes. Después de esa cantidad, dependiendo del plan contratado, existe un coste extra por cada GB adicional.
- Además del acceso a la API, cada plan da acceso a una interfaz web y un plugin de *Wordpress* por si deseamos hacer un uso no programático.
- No existe una capa gratuita, aunque sí se ofrecen 50MB de subida gratis para las pruebas. No se menciona ningún tipo de plan personalizado que se pueda negociar.

4.2.6 Mailgun

- Esta organización, que es parte de *Rackspace*, ofrece un servicio de envío masivo de emails para desarrolladores, es decir, no ofrece otro tipo de interfaz no programática como otros tantos proveedores de servicios de *mailing*. Cada dominio funciona de forma aislada, de forma que si tenemos varios clientes y existe problema de *spamming*, podemos deshabilitar esa cuenta de forma separada. Además, cada email es parseado como JSON, lo que facilita el tratamiento programático. Finalmente, incluye con un servicio de validación y rastreo de emails.

- El modelo de cobro es muy simple, cuenta con una capa gratuita de 10k envíos mensuales y luego un coste incremental por rango de envíos: 500k, 1M, 5M y más. Se puede añadir además una IP dedicada por un coste fijo adicional.
- Existe un plan personalizado donde la propia organización revisa los envíos, monitoriza la reputación de los servidores y garantiza la llegada a la bandeja de entrada. Es preciso contactar con algún asistente de ventas para obtener más información.

4.2.7 OpenWeatherMap

- Esta empresa ofrece información meteorológica, aunque comparte nicho de mercado con otras muchas. Quizás el aspecto más destacable es que se permite que los usuarios con estaciones meteorológicas *amateur* puedan subir sus datos. De esta forma se cubren zonas locales y la predicción no se basa sólo en estaciones oficiales y de aeropuertos.
- De forma análoga a otros competidores, se ofrece una capa gratuita de 3000 llamadas por minuto, aunque los datos tienen un retraso de dos horas y el registro histórico es menor a un mes.
- Cuenta con tres planes basados en el número de peticiones por minuto, aunque también se aumenta el histórico disponible, la disponibilidad garantizada y el soporte del que se dispone.

4.2.8 Semantics3

- Esta compañía ofrece información de productos vendidos en diferentes plataformas de comercio electrónico, de hecho, afirman que almacenan más de 60 millones de productos y de cada uno, las distintas ofertas del mercado. De esta forma se permite acceder a histórico de precios, buscar productos por código de barras, búsqueda semántica de productos, etc.
- Existen 3 planes que cuentan con 1k, 25k y 100k llamadas diarias, además se permite añadir dominios personalizados para monitorizar y crear alertas para cuando existan nuevos productos que encajen con la descripción.
- También cuenta con una capa gratuita de 1k llamadas diarias, pero sin acceso a las imágenes de los productos. Asimismo existen soluciones personalizadas, con un SLA propio y acceso completo a la API.

4.2.9 Stupeflix

- Esta organización ofrece el tratamiento integral de imágenes, vídeo y audio. De esta forma, se puede, por ejemplo, editar un vídeo y subirlo a *Youtube* y *Facebook*, recortar imágenes y crear audio a partir de texto.
- El modelo de cobro es sencillo, no existen planes definidos, sino que cada recurso se tarifica de forma independiente. Además se ofrece un servicio de alojamiento multimedia con un precio fijo por almacenamiento y ancho de banda usados.
- No existe una capa gratuita, aunque sí ofrecen crédito gratuito para las primeras pruebas. Tampoco existe ninguna mención a planes personalizados.

4.2.10 Resumen comparativo

APIs estudiadas								
Plataforma	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
AlchemyAPI	x	x	x	x	x	-	x	x
Algorithmia	-	-	-	x	x	x	-	-
FlightStats	-	-	x	-	x	x	-	-
Groupdocs	x	x	x	-	x	-	-	-
Kraken	-	x	-	-	-	x	-	-
Mailgun	x	-	-	x	x	-	-	-
OpenWeatherMap	x	x	x	-	-	-	-	x
Semantics3	x	x	x	-	x	-	-	x
Stupeflix	-	-	-	x	-	x	-	-

Cuadro 4.1: Resumen comparativo de APIs estudiadas

4.2.11 Análisis comparativo

Cada columna de la tabla §4.1 se corresponde con ciertas características analizadas, de forma que una x en la celda $C_{i,j}$ significa que la plataforma i posee la característica j.

1. Capa gratuita.
2. Varios planes disponibles.
3. Plan basado en un número fijo de peticiones.
4. Plan basado en el número peticiones realizadas.
5. Posibilidad de planes personalizados.
6. Periodo de prueba limitado.
7. Paquetes de soporte adicional.
8. Disponibilidad garantizada por SLA.

4.3 ESTUDIO DE *SaaS*

4.3.1 Dropbox

- Esta conocida empresa ofrece un servicio de almacenamiento de ficheros con numerosas características de valor añadido como son su cliente de sincronización de archivos, el histórico de cambios, la subida automática de fotos desde dispositivos móviles, etc.
- Existen dos segmentos de clientes: los particulares y las empresas. En el primer caso, el modelo de cobro es muy sencillo, por una cuota mensual se consigue 1TB de espacio y ciertas características que el plan gratuito no tiene.
- A nivel empresarial, se tarifica por usuarios que vayan a hacer uso del servicio. Además esta versión cuenta características adicionales enfocadas a mejorar la seguridad y la compartición de archivos.
- La versión gratuita ofrece almacenamiento de 2GB, aunque ampliables según diversas campañas promocionales, y numerosas características básicas.
- Existe una versión de prueba de 14 días de la edición para empresas.

4.3.2 Google Apps

- *Google* ofrece sus aplicaciones más populares como servicio y de manera corporativa. De esta forma, tendremos, por ejemplo, toda la funcionalidad de *Gmail* pero con nuestro propio dominio de la compañía, llamadas y contactos dentro de la empresa, almacenamiento para los documentos editados y compartidos online... en definitiva, todos los servicios de *Google* al servicio de la empresa. Es destacable además la posibilidad de establecer políticas y restricciones de seguridad: incluso se pueden configurar teléfonos *Android* para que estén sujetos a tales políticas.
- Si bien anteriormente existía una versión gratuita limitada, esta ha sido retirada y sustituida por dos planes similares. La diferencia está en que uno tiene almacenamiento ilimitado y registro de emails enviados por la compañía.
- Se tarifica por usuario dado de alta en el servicio y se permite el pago mensual o anual.

- Cabe mencionar aquí que *Google*, en apoyo de organizaciones sin ánimo de lucro y colegios e institutos, ofrece estos servicios de forma gratuita. Existe un límite de usuarios inicial, aunque con una solicitud lo incrementan sin mayor problema.
- Existe una versión de prueba durante 30 días.

4.3.3 Matlab Online

- Desde la compañía Mathworks, creadora del conocido programa *MATLAB*, nos ofrecen una versión en la nube con ciertas limitaciones. Por ejemplo, no podemos ejecutar comandos de más de 5 minutos, acceder al hardware directamente, subir archivos de más de 16MB y otras limitaciones. A pesar de ello, con un ordenador[2] con conexión a Internet y un mínimo de memoria, podemos acceder a este potente software.
- Aquí no existe un modelo de cobro específico, sino que se trata de una solución de valor añadido al adquirir una licencia de *MATLAB*.
- Con una licencia de prueba, podemos disponer de tal servicio durante 30 días.

4.3.4 Office365

- La conocida suite de ofimática busca nuevos nichos de mercado ofreciendo sus productos como servicio. De esta forma, según el segmento de mercado, se crean familias de planes dependiendo de la funcionalidad ofrecida.
- Sin contar planes para ONG, estudiantes y otro tipo de entidades, se ofertan tres tipos de familias de planes: para usuarios domésticos, pequeña empresa y gran empresa.
- Los planes domésticos incluyen *Word*, *Excel*, *PowerPoint* y *OneNote*, 1TB de almacenamiento, 60 minutos de llamadas por Skype. La diferencia entre ellos está en el número de dispositivos en los que se permite la activación.
- En el ámbito empresarial añadimos *Outlook* y *Publisher* y *Access* en algunos planes. La bandeja de entrada de correo electrónico se incrementa a 50GB, se añade una red social corporativa y otras características.

[2] Existe un cliente para dispositivos móviles aún más ligero.

- Salvo en las ediciones personales, se adquiere una licencia por cada usuario, existiendo para algunos planes una limitación de usuarios, a fin de que se adquiera otro tipo de licencia.

- No existen capas gratuitas con funcionalidad reducida ni versiones de prueba.

4.3.5 Sage

- Sage es la compañía detrás de conocidos productos como *Contaplus* o *Facturaplus* que se ha lanzado al mundo del software como servicio. Se ofertan 3 productos: ERP, CRM y un gestor de contabilidad y facturación sencillo. Nos centraremos en *Sage CRM*.

- Existen tres formas de adquirir el software: compra de licencia y despliegue en instalaciones propias, suscripción temporal (igual que el anterior, pero sólo durante un tiempo limitado) y *cloud*.

- En la modalidad *cloud* se ofertan dos versiones, donde la más avanzada ofrece automatizaciones en áreas de marketing, comercial y atención al cliente.

- No hay ninguna capa gratuita con funcionalidad reducida ni versión de prueba.

4.3.6 Salesforce

- Esta compañía fue de las pioneras en la industria *SaaS* del software empresarial. Oferta actualmente 10 productos, cada uno con diferentes planes. Nos centraremos en su CRM *SalesCloud*.

- Existen 5 planes disponibles, consistentes en precio por licencia de uso por usuario y mes. La diferencia entre todas ellas es el número de características que ofrece. En los más avanzados, además, se permite la creación de apps propias y la integración a través de API y se incrementa el soporte.

- No existe ningún plan gratuito, aunque sí una versión de prueba de 30 días. Se ofrece además la posibilidad de combinar las licencias de sus dos servicios más usados *SalesCloud* y *ServiceCloud* por un precio reducido.

4.3.7 Cisco WebEx

- Desde la conocida compañía Cisco llega esta solución corporativa para reuniones, enseñanza a distancia, eventos y soporte. En particular, nos centraremos en *WebEx Meetings*, dedicado a organizar reuniones, compartir contenido y realizar videollamadas de alta calidad.
- Se ofertan 4 planes, que varían en función del número de personas que pueden asistir a la reunión virtual y otras funcionalidades. En el mejor plan, se permiten hasta 100 personas, vídeos de alta definición y 1GB de almacenamiento.
- Se incluye una capa gratuita con reuniones de hasta 3 personas y con funcionalidad reducida.

4.3.8 Zoho

- Bajo el este nombre se sitúan numerosas aplicaciones que podemos agrupar en tres grandes grupos: de negocio, de colaboración y de productividad. Además se facilita la integración con *Google Apps*.
  Puesto que resultaría algo tedioso estudiar todas las aplicaciones detenidamente, comentaremos brevemente aquellas que presentan un mayor interés y algunas generalidades.
- Casi todas las aplicaciones presentan una capa gratuita con la funcionalidad básica, pero con limitaciones de espacio y otros detalles menores. Existen aplicaciones con un plan gratuito y una versión *pro* con todo lo que le falta y otras que sí presentan una mayor granularidad en los planes ofertados.
- En algunas aplicaciones existe también el pago según el volumen de uso y descuentos por tipo de entidad (ONG, escuelas, etc.).
- Esta compañía cuenta además con numerosas aplicaciones móviles para evitar perder el acceso a ellas fuera del lugar de trabajo.

4.3.9 Resumen comparativo

SaaS estudiados				
Plataforma	**1**	**2**	**3**	**4**
Dropbox	x	x	-	x
Google Apps	-	x	-	x
Matlab Online	-	-	x	x
Office365	-	x	x	-
Sage	-	x	x	-
Salesforce	-	x	-	x
Cisco WebEx	-	x	-	-
Zoho	x	x	-	-

Cuadro 4.2: Resumen comparativo de SaaS estudiados

4.3.10 Análisis comparativo

Cada columna de la tabla §4.2 se corresponde con ciertas características analizadas, de forma que una x en la celda $C_{i,j}$ significa que la plataforma i posee la característica j.

1. Capa gratuita.
2. Varios planes.
3. Plan no *SaaS* (se ofrece despliegue *on-premises*, bibliotecas, etc.).
4. Periodo de prueba limitado.
5. Paquetes de soporte adicional.
6. Disponibilidad garantizada por SLA.

4.4 Estudio de API Gateways

[3] Un problema frecuente durante al auge de todas las tecnologías como servicio es cómo obtener un beneficio tras el despliegue de nuestra aplicación. No olvidemos que, en la mayoría de los casos, dar un servicio es sinónimo de aportar valor al cliente o usuario y esto es algo por lo que esperamos un retorno de la inversión concreto. Además, resulta evidente que tras los modelos y líneas de negocio establecidos por la dirección de la empresa existe una base tecnológica que los respalda.

Durante esta sección estudiaremos la base tecnológica de los modelos de cobro y gestión de una API, sin embargo, conviene realizar un breve recorrido sobre los distintos modelos de negocio que se dan con frecuencia en el mercado.

- **Gratis**: se ofrece una capa sin coste alguno para el usuario, de tal forma que éste se familiarice con el uso de la API y sea consciente del valor que entrega. Podemos optar por este modelo si estamos labores de promoción o la API es un servicio de valor añadido para otro producto mayor.
- **Por capas**: se organizan distintos planes que dan acceso a determinados recursos de la API y se facturan de forma independiente.
- **Pay as you go**: se factura de acuerdo al volumen de uso de la API; por ejemplo, teniendo un coste por uso y acumulando éste hasta el final del mes, cuando se realiza el cobro.
- **Modelo personalizado**: para grandes empresas o en caso de necesidades especiales, los proveedores de la API pueden establecer unos modelos de cobro individualizados con fines estratégicos o económicos.

Conviene ahora ver qué se está usando en el mercado para resolver los asuntos derivados de la gestión de una API. En concreto, dedicaremos esta sección a hablar de los *API gateways* con mayor popularidad actualmente, pasando por las ventajas que ofrecen y comparando entre las distintas alternativas.

En un intento de analizar la oferta de proveedores de *API gateways* que existe actualmente en el mercado hemos logrado reunir información [4] de más de 50 empresas

[3] A partir de esta sección se ha elaborado un artículo que será presentado en las *XI JORNADAS DE CIENCIA E INGENIERÍA DE SERVICIOS*; en el anexo §11.2 se muestra tal texto.

[4] Gran parte de esta búsqueda ha sido realizada por Kin Lane.

y proyectos con una temática relacionada. No obstante, a fin de destacar y realizar una comparación de mayor granularidad, hemos seleccionado 13 proveedores:

1. 3Scale.
2. Akana API Gateway.
3. API Umbrella.
4. Apiaxle.
5. Apigee Edge API.
6. Axway API Gateway.
7. Azure API Management.
8. CA API Gateway.
9. Mashape.
10. Mashery API Gateway.
11. Monarch API Manager.
12. Repose.
13. WSO2 API Management.

Entre las características esperables para este tipo de sistemas se incluyen algunas como analíticas, autenticación autorización, cifrado, filtrado, limitación de peticiones y tratamiento de respuestas.

En base a todo esto, analizaremos cada uno de ellos y tabularemos los resultados obtenidos. De esta forma tendremos una base sólida sobre la oferta actual del mercado en cuanto a servicios de gestión y control de APIs.

4.4.1 3Scale

- Aunque existen varios planes no personalizados, si vemos el más completo, podemos tener hasta 3 APIs con tráfico diario de un millón de llamadas y con hasta 5000 desarrolladores registrados. Además se pueden establecer cuotas por usuario final.
- En cuanto a control de acceso y seguridad de la API, da soporte a *API keys*, control por desarrollador y aplicación, alertas al violar los límites establecidos, *OAuth* y lista blanca/negra de IP.
- En relación a las cuotas y políticas de tráfico, se permite crear planes personalizados que tengan diferentes funcionalidades y límites.
- Para realizar analíticas, se muestrean las llamadas realizadas, tanto por usuario como por aplicación.

- Se permiten además planes basados en el número de peticiones, de modo que el cobro sea en función del uso.
- Se genera un portal para desarrolladores, personalizable y con documentación basada en *Swagger*.
- Presenta un modo de despliegue en sus servidores (*APICast*) o directamente recae sobre un proxy inverso con *Nginx*, *Varnish* o proveedores *cloud* como *AWS* o *Heroku*.
- Finamente, cabe destacar aquí la sencillez con la que un desarrollador puede poner en marcha el servicio con una API existente; la capa gratuita permite un uso de la API con hasta 50k llamadas diarias.

4.4.2 Akana API Gateway

- Esta compañía cuenta con un catálogo de soluciones y productos orientados a SOA, por lo que la administración de APIs es sólo una de sus líneas de trabajo.
- En cuanto a control de acceso y seguridad de la API, da soporte a *API keys*, control por usuario y aplicación y *OAuth*. Además mediante cifrado XML y soporte SSL/TLS se incrementa la seguridad de la API sin tener que añadir código específico. Es destacable también la prevención contra DoS, mensajes malformados o con demasiada profundidad, además de detectar inyecciones SQL, XPath/XQuery...
- Nos ofrece además una transformación bidireccional entre mensajes SOAP/XML vía JMS y APIs RESTful mediante JSON/XML. Permite también orquestación de APIs como si de una governanza SOA clásica se tratase.
- En temas de monitorización cuenta con análisis en tiempo real de las llamadas realizadas con sistema de alertas, todo ello para asegurar el cumplimiento de los SLA.
- El despliegue se puede realizar tanto *on-premises* en los servidores propios como en proveedores *cloud* externos.

4.4.3 API Umbrella

- Es un proyecto *Open Source* que nos ofrece un proxy inverso para nuestra API. No presenta características tan avanzadas como otras soluciones analizadas.

- Se realiza un balanceo de carga inicial con *Nginx* y las peticiones que vayan dirigidas a la API serán redirigidas al servidor con *NodeJS*, mientras que las que vayan al portal de desarrolladores serán dirigidas al servidor con *Ruby on Rails*.
- Sobre las peticiones realizadas a la API, permite autenticación de peticiones por *API key* y roles, limitación del tráfico y reescrituras de respuestas. Asimismo, mediante *Varnish*, se realiza un cacheado de las respuestas. Finalmente las peticiones son enrutadas a los diferentes API backends de los que dispongamos.

4.4.4 Apiaxle

- Esta es otra herramienta *Open Source* que se vuelve a basar en *Nginx* y *Varnish* para realizar sus funciones.
- Permite autenticación por *API Keys*, limitación del tráfico de peticiones y detección de errores en JSON/XML.
- Cuenta con estadísticas en tiempo real de los distintos eventos que se suceden.

4.4.5 Apigee Edge API

- Esta compañía ha desatollado una herramienta comercial con un modelo de negocio por uso, si se opta por el despliegue en sus servidores. Si se prefiere un sistema *on-premises*, se adquieren licencias por servidor.

  En cuanto a control de acceso y seguridad de la API, permite especificar numerosos sistemas de autenticación, desde el simple *API key* hasta mecanismos propios, pasando por *OAuth* y *SAML*. También se permiten establecer cuotas y límites por aplicación y usuario. Es destacable además la prevención contra ataques de seguridad según las distintas políticas y *workflows* que presenta integrados.
- De cara a los desarrolladores que usen la API, se permiten establecer distintos planes con sus cuotas de uso específicas. Además se pueden realizar paquetes con las APIs que deseemos exponer y extraer variables desde la respuesta, monitorizarlas y tenerlas en cuenta en los planes.
- Entre otras características destacan la posibilidad de realizar versionado de la API, las transformaciones de protocolo sobre la respuesta de la API y el sistema de caché y de distribución de contenido distribuido.

- En relación a formas de cobro que existen, podemos personalizar el *workflow* de acuerdo a nuestras necesidades y, mediante el portal para desarrolladores, se puede centralizar y controlar todo el proceso.
- El sistema de monitorización a tiempo real nos permite establecer alertas y reportes tanto a nivel técnico como de de negocio.

4.4.6 Axway API Gateway

- Esta compañía cuenta con un fuerte trasfondo de SOA y la mayoría de sus productos están orientados a ello. En este caso, pretenden dotar a las APIs de la misma idea de governanza clásica.
- Se pretende controlar el ciclo de vida del desarrollo de APIs, disponiendo además de un catálogo donde se publiquen. Se incluyen políticas integradas para facilitar la configuración y desarrollo.
- En cuanto a control de acceso y seguridad de la API, esta solución cumple numerosos estándares de seguridad y nos permite emplear mecanismos de autenticación propios, además de conectores con conocidos productos de IBM y Oracle y soporte a *OAuth*, *LDAP*, etc.
- La monitorización a tiempo real de las peticiones a la API permite controlar el cumplimiento de SLAs y QoS. Además ofrece planes configurables y cuotas de uso.
- Entre otras características se encuentra el balanceo de carga, versionado, cacheado de peticiones y conversión de respuestas SOAP/REST.
- En el sentido de la governanza SOA, da soporte a la recepción de mensajes asíncronos (*JMS*, *Websockets*...) y orquestación de servicios.

4.4.7 Azure API Management

- Entre los servicios que ofrece Microsoft en su solución *cloud* se incluye la administración de APIs desplegadas en sus sistemas.
- En cuanto a control de acceso y seguridad de la API, se pueden establecer cuotas y límites de uso. La autenticación se realiza de forma nativa, según los propios

mecanismos de esta; no obstante, se pueden aplicar políticas de grupo por *Active Directory*.

Además, ofrece un portal para desarrolladores con consola para realizar pruebas, un sistema de monitorización y alertas y permite también la conversión XML/JSON de las respuestas.

4.4.8 CA API Gateway

- Es un producto de una empresa que también prioriza la governanza SOA en su catálogo de servicios. En particular esta solución pretende servir de soporte a aplicaciones desarrolladas para dispositivos móviles.
- En cuanto a control de acceso y seguridad de la API, se da soporte a múltiples mecanismos de autenticación, incluyendo *LDAP*, *SAML*, *OAuth* y *Single Sign On (SSO)*. Se pueden establecer límites y métricas de cara al cumplimiento de SLAs. Además pretende proteger la API frente diversos ataques.
- Entre otras características, ofrece cacheado de peticiones, versionado y analíticas.

4.4.9 Mashape

- Además de ofrecer los servicios de *gateway*, se introduce el concepto de *marketplace*, donde a través de una interfaz similar se pueden consumir las APIs publicadas. Es una organización orientada a la sencillez y facilidad de uso de cara al desarrollador.
- En cuanto a control de acceso y seguridad de la API, se acepta autenticación básica y *OAuth*. Además se pueden enviar *headers* exclusivos entre el proxy y la API, de forma que la petición quede transparente para el desarrollador.
- Se pueden crear planes basados en peticiones y *endpoints*, además de poder establecer hasta cuatro cuotas distintas De esta forma, con un *HTTP header* podemos controlar las unidades que se incrementa hasta llegar al límite. Asimismo, se pueden crear planes privados sólo para ciertos desarrolladores invitados.
- La monitorización permite verificar el estado de la API y generar alertas, además de visualizar los ingresos generados y cuotas usadas por los distintos desarrolladores.

- Permite crear un portal con documentación y consola para pruebas, además de generar código en varios lenguajes para consumir rápidamente la API. Permite también ofrecer soporte a los usuarios con un sistema de tickets.

4.4.10 Mashery API Gateway

- Mashery fue una de las primeras organizaciones en adoptar el término *API management*, aunque también por ello cuenta con una funcionalidad más limitada.

- Los mecanismos de autenticación son básicos, por IP o mediante autenticación HTTP. De cara al control, se permite establecer límites de uso, pero de una forma básica, accediendo directamente al portal que ofrece.

- También se pueden realizar analíticas y recopilar estadísticas del uso de la API.

4.4.11 Monarch API Manager

- Es una ligera herramiente *Open Source* que se puede desplegar *on-premises* o en *cloud* de dos formas distintas: directamente en el código de la API o mediante un proxy inverso con *Nginx*.

- Permite establecer políticas básicas de seguridad basadas en *OAuth* a fin de gestionar el control de acceso a la API. Además realiza una monitorización a tiempo real que almacena sobre *MongoDB*.

4.4.12 Repose

- Es otra solución Open Source *middleware* para la gestión de APIs que se despliega *on-premises* y permite que sólo peticiones bien formadas y validadas lleguen a la API a través de un sencillo *workflow*:

- En primer lugar, se realiza la autenticación por *API Key*, HTTP o IP y se comprueba que no se ha sido superado el límite de peticiones. En caso de autorizarse, se realiza un balanceo de carga y se sirve la petición donde corresponda. Finalmente, se guarda el registro de la petición en el log.

4.4.13 WSO2 API Management

- Como todos los productos de WSO2, es una herramienta *Open Source* desplegada *on-premises* que permite realizar una gestión de las APIs durante todo el ciclo de vida.
- En cuanto a control de acceso y seguridad de la API, se permiten mecanismos de autorización como *OAuth*, *LDAP* y *SAML*. Además se pueden establecer planes y límites en el uso de la API.
- Cuenta una un panel de administración donde controlar el estado de publicación de cada API y un portal con la documentación especificada con *Swagger*.
- Realiza monitorización de las peticiones realizadas para ofrecer reportes, generar alertas y garantizar el cumplimiento de los SLA.

4.4.14 Análisis comparativo

Cada columna de la tabla §4.3 se corresponde con ciertas características analizadas, de forma que una x en la celda $C_{i,j}$ significa que la plataforma i posee la característica j.

Es importante destacar que la ausencia de esa marca no implica directamente que la plataforma no posea la característica, es posible que no lo publiciten en sus especificaciones y luego esté incluida.

1. *Open Source*.
2. *Free tier*.
3. Autenticación básica.
4. Autenticación avanzada.
5. Planes personalizados.
6. Cuotas por uso.
7. Analíticas.
8. Portal de desarrolladores.
9. Soporte al versionado.
10. Conversión de respuestas.
11. Protección contra ataques.
12. Balanceo de carga.
13. Caché de peticiones.
14. Orquestación de servicios REST.

API *Gateways* estudiados														
Plataforma	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
3Scale	-	x	x	-	x	x	x	x	-	-	-	-	-	-
Akana API Gateway	-	-	x	x	-	x	x	-	-	x	x	-	-	x
API Umbrella	x	x	x	-	-	-	x	x	-	-	-	x	x	-
Apiaxle	x	x	x	-	-	-	x	-	-	-	-	-	-	-
Apigee Edge API	-	-	x	x	x	x	x	x	x	-	x	-	x	-
Axway API Gateway	-	-	x	x	x	x	x	-	x	x	-	x	x	x
Azure API Management	-	-	-	-	-	x	x	x	-	x	-	-	-	-
CA API Gateway	-	-	x	x	-	-	x	-	x	-	x	-	x	-
Mashape	-	x	x	-	x	x	x	x	-	-	-	-	-	-
Mashery API Gateway	-	-	x	-	-	x	x	x	-	-	-	-	-	-
Monarch API Manager	x	x	x	-	-	-	x	-	-	-	-	-	-	-
Repose	x	x	x	-	-	x	x	-	-	-	-	x	-	-
WSO2 API Management	x	x	x	x	x	x	x	x	x	x	-	-	-	-

Cuadro 4.3: Resumen comparativo de API *gateways*.

Si aplicamos técnicas propias del *análisis formal de conceptos* (completo en C.1), podemos ver fácilmente las tendencias del mercado:

- Todos ofrecen analíticas sobre el uso de la API.
- La mayor parte da soporte a mecanismos de autenticación básicos.
- Los que ofrecen versionado o protección suelen permitir mecanismos de autenticación más avanzados.
- Casi todos los que permiten cuotas y y planes ofrecen un portal para desarrolladores.

4.4.15 Estudio de Mashape

Como ya hemos visto en §4.4.9 de la sección anterior *Mashape* nos ofrece los servicios de *gateway* para nuestras APIs y además un *marketplace* donde publicarlas. Al estar orientado al pequeño desarrollador y no a grandes organizaciones enfocadas a SOA, la sencillez con la que podemos publicar una API existente es bastante superior a la del resto de proveedores estudiados en la sección anterior. Justo por eso dedicaremos esta sección a detallar las funcionalidades que nos ofrece a través de un ejemplo de uso. Aunque bien podríamos hablar de otros proveedores de soluciones *Open Source* o con una capa gratuita, puesto que la funcionalidad ofrecida es muy similar no merece detenerse tanto.

Partiremos de una API existente desplegada en el *endpoint* siguiente: HTTP://PAPAMOSCAS-ISA.APPSPOT.COM/API/V3. En primer lugar debemos asociar nuestro *endpoint* con el proxy inverso de Mashape; esto es inmediato, pues desde un simple formulario web obtenemos la dirección detrás del proxy: HTTP://PAPAMOSCAS.P.MASHAPE.COM. En la figura §4.1 se puede ver este proceso.

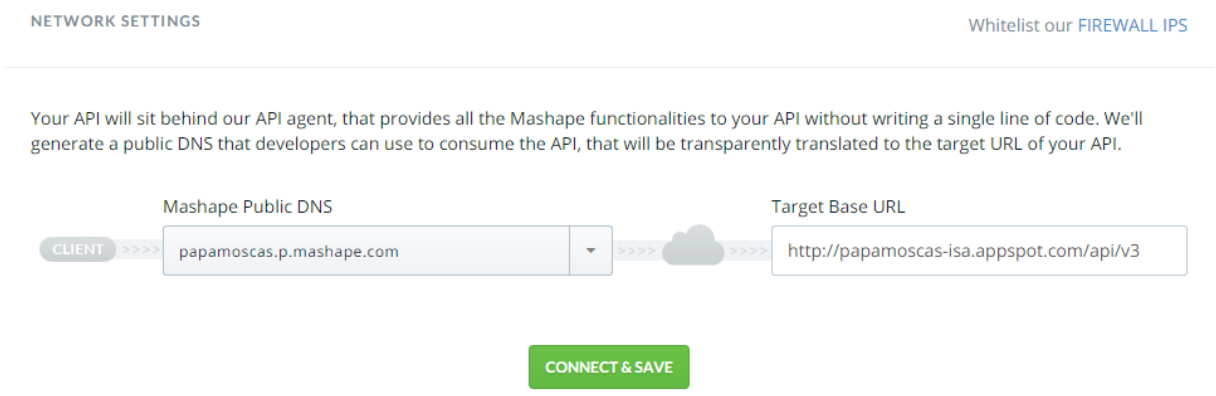

Figura 4.1: Configuración del proxy en *Mashape*.

Una vez que hemos obtenido la dirección pública de nuestra API deberíamos adaptar nuestra API para que sólo permitiese peticiones si se envía un *header* del tipo X-MASHAPE-PROXY-SECRET: XXXXXX, pero como la API desplegada es compartida por otros ejemplos de la presente memoria, hemos optado por omitir tal cambio. Es interesante destacar que podemos enviar *headers* entre el proxy de *Mashape* y nuestra API, así podemos realizar diversas acciones programáticamente.

Una vez definido el *endpoint* de *Mashape* empezamos a definir los diferentes re-

cursos que la API ofrece. De esta forma estamos haciendo el mapeo entre el proxy y nuestra API y a la vez generamos una documentación y entorno de pruebas para los desarrolladores.

Debido al diseño de *Mashape*, es preciso que los desarrolladores registren una *Aplicación* que consume la API. De esta forma se puede controlar y limitar las peticiones de los diferentes planes que luego crearemos. Esto es independiente de la autenticación de usuarios dentro de la propia API, como ahora veremos.

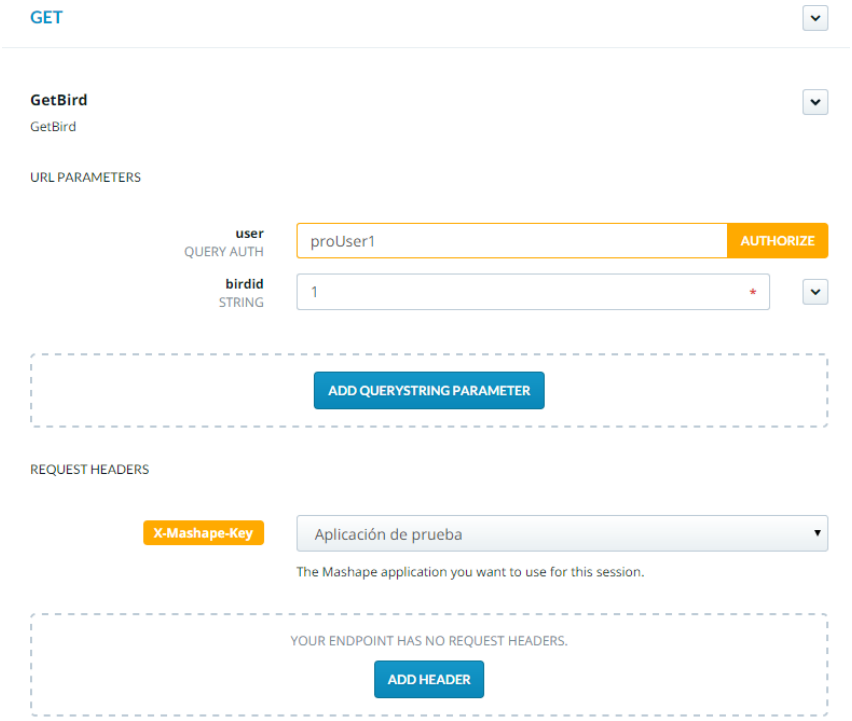

Figura 4.2: Creación de los diferentes recursos en *Mashape*.

Como se aprecia en las figuras §4.2 y §4.3 el proceso es bastante intuitivo y sólo requiere conocer los datos que usa nuestra API.

Respecto a autorización de peticiones en *Mashape* hay que distinguir la autenticación propia que use nuestra API y el registro de la aplicación de *Mashape* que hace uso de nuestra API (y cualquier otra). Los mecanismos de autenticación soportados son:

- *Basic Authentication*.
- *Header Authentication*.

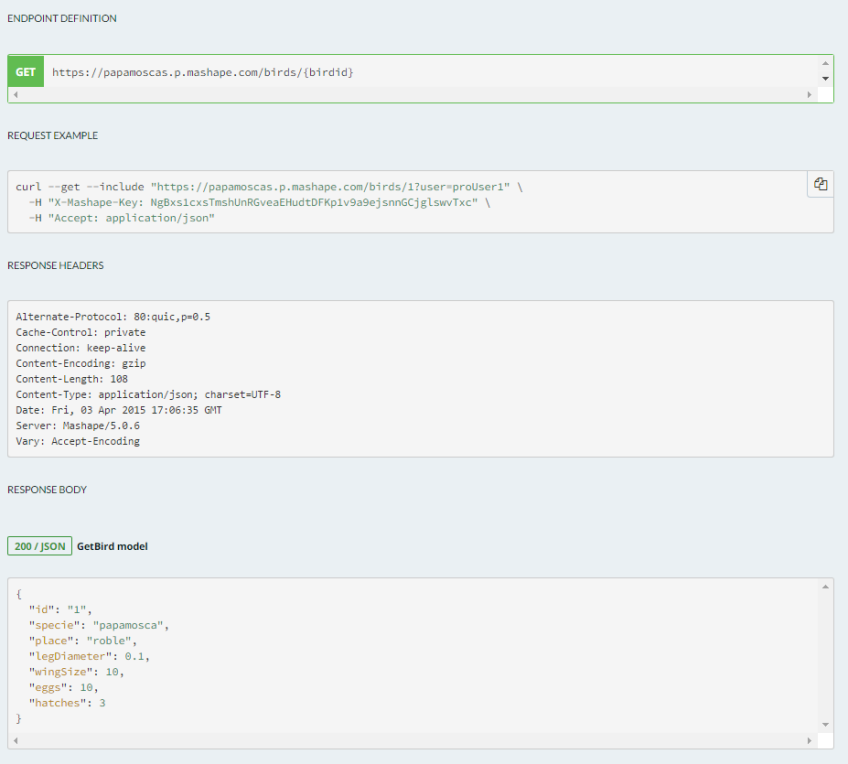

Figura 4.3: Creación de los diferentes recursos en *Mashape*.

- *Query Authentication*.
- *OAuth 1 Authentication*.
- *OAuth 2 Authentication*.

En primer lugar, la API que usamos, emplea una autenticación basada en *Query Authentication*, esto es, el *endpoint* requiere el par ?USER=TOKEN. De ahí que en §4.2 aparezca PROUSER1.

Hasta aquí el proceso de definición de los diferentes recursos que nuestra API ofrece, el siguiente paso es definir los planes para cobrar por el uso de la misma.
Sólo se pueden establecer cuatro planes: *basic*, *pro*, *ultra* y *mega*, aunque se pueden establecer planes privados e invitar a desarrolladores elegidos a ellos.

Como se aprecia en la figura §4.4, cada plan puede tener cuatro tipos de *cuotas* distintas y diferentes *características*. Actualmente, *Mashape* sólo tiene en cuenta las cuotas. Por defecto, cada llamada al *endpoint* especificado en la cuota hará que esta se incremente en una unidad. No obstante, este comportamiento se puede modificar mediante *headers*: X-MASHAPE-BILLING: [QUOTA] = [VALUE] (;[QUOTA] = [VALUE]).

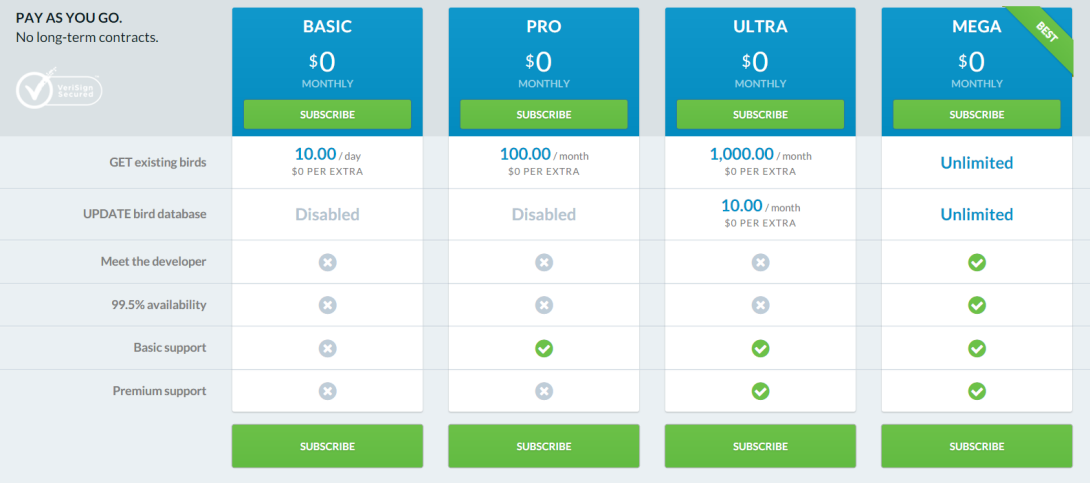

Figura 4.4: Creación de planes en *Mashape*.

Si se quiere tratar las *características* de forma programática en la API, *Mashape* genera, entre otros[5], un *header* (X-MASHAPE-SUBSCRIPTION) para indicar el nombre del plan.

Desde el punto de vista del consumidor de la API, el proceso es aún más simple; basta con crear una *aplicación* que haga uso de una o varias APIs y crear la *key* que enviaremos al realizar la petición a la API como un *header* (X-MASHAPE-KEY).
Según el plan al que nos hayamos subscrito, si superamos las peticiones máximas, se nos bloqueará el acceso a la API o se cobrará en función de las peticiones extra.

Con esto, ya podemos usar la API desde otro lugar, basta con hacer una petición como 4.1.

```
1  GET /birds?user=proUser1 HTTP/1.1
2  Host: papamoscas.p.mashape.com
3  Connection: keep-alive
4  X-Mashape-Key: NgBxs1cxsTmshUnRGveaEHudtDFKp1v9a9ejsnnGCjglswvTxc
5  CSP: active
6  User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64)
      ↪ AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2353.0
      ↪ Safari/537.36
7  Accept: */ *
8  Accept-Encoding: gzip, deflate, sdch
9  Accept-Language: es,en;q=0.8
10 Cookie: ajs_anonymous_id=%22d64c0f0f-202e-4040-
      ↪ b336-f5023ca84555%22; _cio=e8ecbc7a-9880-
```

[5]El resto se pueden consultar en la documentación

```
    ↪ bada-0fa0-aa28117c9a18; _cioid=53aa2ddae4b051a76d23d300;
    ↪ _ga=GA1.2.882267961.1428064410; ajs_group_id=null;
    ↪ ajs_user_id=%2253aa2ddae4b051a76d23d300%22
```

Código 4.1: Petición a la API a través de *Mashape*.

Y obtenemos una respuesta como la que se muestra en 4.2.

```
1  HTTP/1.1 200 OK
2  Alternate-Protocol: 80:quic,p=0.5
3  Cache-Control: private
4  Content-Encoding: gzip
5  Content-Type: application/json; charset=UTF-8
6  Date: Fri, 03 Apr 2015 15:52:44 GMT
7  Server: Mashape/5.0.6
8  Vary: Accept-Encoding
9  Content-Length: 164
10 Connection: keep-alive
11
12 [
13   {
14     "id": "1",
15     "specie": "papamosca",
16     "place": "roble",
17     "legDiameter": 0.1,
18     "wingSize": 10,
19     "eggs": 10,
20     "hatches": 3
21   }
22 ]
```

Código 4.2: Respuesta de la API a través de *Mashape*.

Lo último que cabe mencionar son las analíticas que *Mashape* realiza; si bien son muy simples, nos pueden servir para tener una primera impresión del estado y salud de la API, así como del uso medio que realiza cada desarrollador subscrito. En la figura §4.5 se puede ver un ejemplo de la salida que nos ofrece.

Mashape ofrece además otros planes de pago tanto para desarrolladores como para publicadores de APIs. La diferencia con los gratuitos radica en el soporte, las comisiones en cada pago y en las analíticas que se ofrecen.

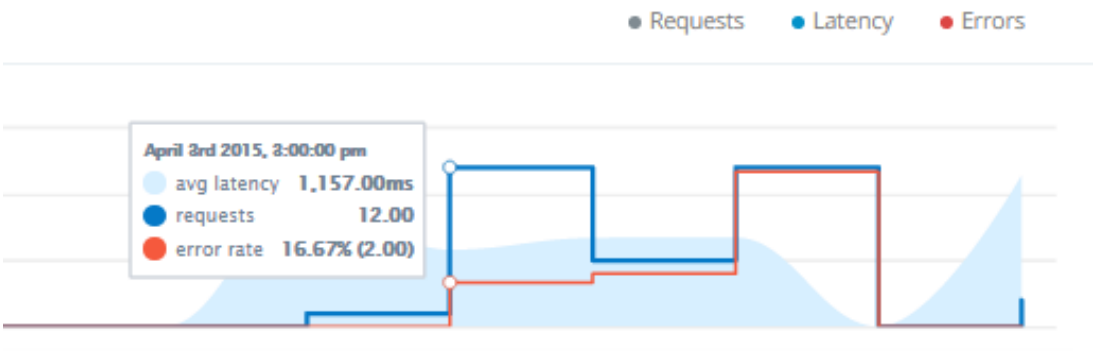

Figura 4.5: Monitorización de peticiones en *Mashape*.

En conclusión, podemos decir que *Mashape* ofrece un buen servicio de *gateway* de API, posibilitando a los usuarios tener sus APIs centralizadas y sin preocuparse por la existencia de planes en sus desarrollos. Sin embargo, las limitaciones que ofrece son evidentes: sólo cuatro tipos de planes, cuatro cuotas monitorizables, analíticas escasas; y sobre todo, la comisión por venta se sitúa en el 20 por ciento, algo que puede hacernos perder bastantes beneficios.

Estudio tecnológico

Life is like a piano. What you get out of it depends on how you play it.

Tom Lehrer (1928),
American singer and mathematician

Realizaremos un vistazo general al mundo de los frameworks Java, para centrarnos de lleno en dos: JavaEE y Spring. Los analizaremos y debatiremos, mediante pequeñas pruebas de concepto, el mejor de cara al objetivo de este proyecto.

5.1 Evaluación JavaEE vs Spring

Es un hecho que una gran parte de las aplicaciones de carácter empresarial están escritas en **Java**; si bien no hay que olvidar que otros desarrolladores eligen otros lenguajes y plataformas para desarrollar sus proyectos. Sin embargo, el principal foco de usuarios al que va dirigido esta biblioteca está claro: empresas con aplicaciones web con un *backend* Java y con necesidades de integrar los SLA en su lógica de negocio. Por ello, partiremos de la base que usaremos este lenguaje para diseñar nuestra biblioteca.

Justamente en esta línea, adelantamos que usaremos **Maven** como herramienta para la construcción y gestión de proyectos Java. Con esta herramienta, se simplifica enormemente la labor de empaquetar nuestro proyecto como un simple JAR que pueda ser importado como una dependencia en el proyecto que desee usar nuestra biblioteca.

Partiendo de esta línea base, ya solo nos queda entrar en otro complejo y polémico terreno: el mundo de los *frameworks*.

Como adelantábamos arriba, la decisión de elegir o no un *framework* y, en caso de usarlo, cuál emplear, es algo difícil y compleja. Es por ello por lo que dedicamos esta sección a discutir sobre la mejor opción para nuestro proyecto.

En secciones anteriores hemos hablado de las metas de este proyecto, por lo que es evidente la necesidad de una fácil integración, modularidad y escalabilidad. Justamente estas son las ventajas de usar un *framework* para esta labor. No queremos reinventar la rueda. No queremos un proyecto donde demostrar el conocimiento de un lenguaje. Estamos buscando algo real, algo que aporte valor al proyecto que decida emplear nuestra biblioteca. Además, los proyectos reales emplean *frameworks* para obtener un desarrollo más eficiente, seguro y a menor coste.

Si realizamos un recorrido por el vertiginoso mundo de los *frameworks* y, más concretamente, para aplicaciones web, podemos ver observar un *timeline* como el siguiente:

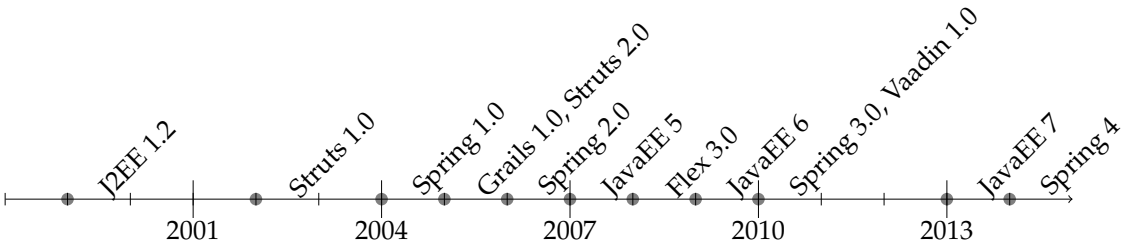

Figura 5.1: Cronología de diversos *frameworks*

En un rápido vistazo podemos observar cómo ha sido la evolución de todos estos *frameworks* para desarrollo de aplicaciones Java. Además del notable incremento de ellos y sus versiones, es destacable el dualismo JavaEE-Spring, aunque posteriormente profundizaremos sobre ello.

Con el evidente auge de las aplicaciones web, los *frameworks* orientados al *modelo-vista-controlador* (MVC) han ido en un claro aumento. Sin embargo, nuestro proyecto no se enmarca plenamente en este patrón de diseño, por lo que debemos buscar algo más genérico y potente.

De entrada, vamos a descartar Grails, Vaadin, GWT, JSF, Struts y Play. Hay una mezcla de motivos para ello: puede que no cuadren con el enfoque del proyecto, algunos son demasiado nuevos, el porcentaje de uso en la comunidad no es demasiado alto y falta conocimiento técnico sobre ellos para poder abordar el proyecto con éxito. Ya hemos comentado anteriormente que no queremos que esto quede en un mero prototipo, una excusa para aprender una nueva tecnología... deseamos que esta biblioteca llegue a ser usada en proyectos reales y que, realmente, aporte valor.

Habiendo eliminado tantos competidores en el párrafo anterior, nos vemos obligados a volver a realizar la polémica comparativa entre el *stack* JavaEE y Spring. Además, es importante destacar que lo haremos en sus últimas versiones, a la hora de escribir esta memoria: **Spring 4.1** y **JavaEE 7**. Es bien conocido por todos que cada uno ha tenido sus buenos y malos momentos a lo largo de la historia: no es justo comparar Spring 4 con J2EE 1.4, por ejemplo.
Sin embargo, antes de empezar a comparar, conviene echar la vista atrás y ver la historia que hay detrás de ambos *frameworks*.

5.1.1 Historia

En 1999, *Sun Microsystems* saca a la luz un nuevo y prometedor *framework*: J2EE 1.2. Los *Enterprise Java Beans* (EJB) pretendían mejorar lo que apenas 3 años atrás se habría construido: el lenguaje Java.

Trabajar con EJB 1.1 era una ardua tarea: estilo de programación verboso, muchas tareas repetitivas, una enorme cantidad de clases, interfaces y archivos XML...
Justo por ese motivo, en 2002, Rod Johnson propone una solución distinta: hacer todo lo que hacía EJB (2.0), pero de una forma más liviana; lo llamaría *Spring Beans*.

En 2004, Spring ya es una alternativa real a J2EE. Ofrece lo mismo que J2EE, además de la compatibilidad con *frameworks* de acceso a datos (JDBC, Hibernate, iBatis) y algo mucho más interesante: inversión del control e inyección de dependencias.

En 2006 la *Java Community Process* (JCP): lanza JavaEE 5 y convierte muchos productos de la comunidad en especificaciones (i.e. Hibernate) y añade algo realmente interesante: las anotaciones. Ya no son necesarios tantos descriptores XML para configurar la aplicación.

Rápidamente *SpringSource*, la empresa que estaba detrás de Spring, responde: añaden soporte a anotaciones a su inyector de dependencias y mejoran lo que ofrecía JavaEE. Por ejemplo, EJB 3.0 sólo permitía inyecciones dentro de un *bean* EJB.

Ya en 2009, la JCP lanza JavaEE 6, saliendo así a la luz CDI (contextos e inyección de dependencias). Ya no es necesario un contenedor de EJB y hay un manejador de eventos e interceptores mucho más potente (rivalizando así con la programación orientada a aspectos de la que tanto presumía Spring). Además añade soporte a otras sub-especificaciones, como JAX-RS para hacer servicios web REST. En este momento, el SpringSource es adquirida por VMware y Rod Johnson abandona el proyecto.

En 2013 sale a la luz JavaEE 7, añadiendo funcionalidades relacionadas con la productividad (i.e. JMS, *batch processing*, caché, concurrencia, validación de *beans*) y con soporte a tecnologías web más recientes (i.e. HTML5, *Websockets*, JSON).

Finalmente, en 2014, Spring también responde con nuevas novedades: soporte a Java 8, se basa en la especificación de *Servlet* 3.0, añade soporte a *Websockets*, mensajería y otras mejoras menores.

Tras este pequeño vistazo de la vida de JavaEE y Spring, conviene destacar un concepto muy importante que los diferencia completamente: mientras que Spring sí puede

ser considerado un *framework* como tal, JavaEE es una especificación. Esto es, Spring es un conjunto de bibliotecas descargables desde la página oficial del proyecto o de los repositorios centrales de Maven, mientras que Java EE es un conjunto de especificaciones estándares, escritas, revisadas y aprobadas por una comunidad de expertos que definen qué servicios debería proveer y cómo se debería programar en una plataforma para aplicaciones empresariales. Además, suelen ofrecer implementaciones de referencia de muchas de las especificaciones.
Por ejemplo la de JPA 2 es EclipseLink (otras son Hibernate de Red Hat, TopLink de Oracle y OpenJPA de Apache). La implementación de referencia de EJB 3.1 es el contenedor de EJBs de Glassfish (otras pueden ser JBoss de Red Hat, Weblogic de Oracle, Websphere de IBM o TomEE de Apache) y la implementación de referencia de CDI 1.0 es Weld de Red Hat.

Y estamos llegando a un punto más diferenciador aún: la portabilidad. JavaEE es el conjunto de especificaciones que define los servicios que debe proveer un servidor de aplicaciones. Por ejemplo, un EJB se usa y configura de la misma forma en JBoss, Weblogic, TomEE o Glassfish. De tener algún problema con alguno, podemos migrar a otro sin necesidad de modificar el código de nuestra aplicación. Sin embargo, es imprescindible que el servidor destino implemente la misma versión de la especificación que el servidor origen.
Sin embargo, con Spring, esto no es así: el propio contenedor de dependencias está integrado en esta biblioteca, por lo que podemos desplegar en cualquier servidor, por ejemplo, Tomcat. Esta portabilidad se paga, y muy caro: el tamaño del WAR generado es excesivamente grande comparado con el de un proyecto en JavaEE.
Nótese entonces que con Spring estamos ligados a una sola compañía, actualmente, VMware. Si la compañía abandona el proyecto o la comunidad se disuelve, estamos ante una situación bastante inquietante.
Parece evidente que, en esta situación, empresas que busquen estabilidad, confiabilidad y alta disponibilidad no dudarán en apostar por JavaEE, mientras que en otros desarrollos donde se persigan otras metas, Spring puede ser un gran aliado.

No obstante y, a pesar de la visión beligerante que hemos ofrecido en párrafos anteriores, hay que mencionar que Spring y JavaEE pueden coexistir, de forma que los *gaps* de uno, los ocupe otro. Spring tiene una estructura modular (i.e. Spring Core, Spring Security, Spring WS, Spring LDAP...), por lo que podemos integrarlos en un proyecto JavaEE sin problemas. Además, los *Spring Beans* pueden inyectarse dentro de los *beans* de JSF y EJB.
En definitiva, no hay ninguna postura clara hoy en día. Cada plataforma ofrece sus

ventajas e inconvenientes. El conflicto por la hegemonía en desarrollo Java empresarial sigue abierta.

5.1.2 Integración de bibliotecas

Tras haber hecho un recorrido por la historia de Spring y JavaEE, merece la pena centrarnos y reflexionar sobre qué es lo que más nos interesa para diseñar nuestra biblioteca.
Para hacer una decisión más fundamentada, podríamos diseñar una pequeña biblioteca de prueba, con una funcionalidad extremadamente simple, que nos sirva de prueba de concepto que soporte nuestra decisión final. Por ejemplo, vamos a implementar una biblioteca que nos ofrezca la posibilidad de guardar pares *Objeto, Metadatos*, posteriormente, exportaremos dicha biblioteca como un JAR que será usado mediante dependencia Maven en otro proyecto. Sin duda, algo muy sencillo, pero nos servirá de base para poder decidir.

Antes de empezar, conviene comentar que, dado que apostamos por garantizar la máxima portabilidad posible (incluso si el despliegue se realiza en servicios de *Platform as a Service*, PaaS), realizaremos el despliegue en un servidor local Tomcat 7. Como es de esperar, habrá que añadir las dependencias manualmente para que el contenedor de EJB o CDI corra en el servidor.

Biblioteca en JavaEE - Proyecto principal en JavaEE

Si apostamos por una biblioteca realizada en JavaEE 7, lo primero que tenemos que hacer es elegir si nuestro *bean* será de EJB o CDI. En nuestro caso, vamos a optar por CDI, pues, para nuestro simple objetivo, no existe diferencia alguna.
El código es bastante sencillo: se trata de un *singleton* que tiene un atributo *Map<Object, Map<String, Object>>*, que será la estructura de datos necesaria para mantener los metadatos asociados a cada objeto. Se puede ver completo en el C.2 del anexo.

Como acabamos de ver, es bastante sencillo. Ahora llega el siguiente paso, empaquetar el proyecto como un JAR e importarlo en el proyecto principal. Para ello, sólo debemos editar el POM de nuestra biblioteca, añadiendo la información de la versión, *groupId*, *artifactId* y en *packaging*, elegir JAR. Con ello, al ejecutar en Maven el comando *install-file*, tendremos instalado en nuestro repositorio local el JAR de la biblioteca.

A continuación sólo nos queda crear un simple proyecto que use esta biblioteca que acabamos de crear. En particular, queremos una aplicación web que muestre un objeto

con los metadatos asociados.

Para ello, habiendo creado ya un proyecto Maven, vamos a introducir las dependencias necesarias en el POM. En concreto, añadimos nuestra biblioteca que acabamos de instalar en el repositorio local y un contenedor de CDI, tal y como comentamos anteriormente. Vamos a usar la implementación de referencia de CDI, Weld de Red Hat. En el fragmento C.3 del anexo, vemos el extracto del POM.

Con esto hecho, definimos un nuevo *servlet* que escuche las peticiones en y muestre la cadena dada por un método de un *bean*. Esto es trivial, tal y como se aprecia en el C.4 del anexo.

Y ahora viene un pequeño problema con el que nos encontramos: CDI escanea los *beans* de nuestro proyecto por defecto. Si queremos incorporar otros, basta con añadirlos al archivo *beans.xml*. Ahora bien, ¿qué ocurre si se actualiza el JAR? Evidentemente, no se actualizará. Por este motivo, buscamos otra solución alternativa.

Simplemente creamos una clase con un método productor que nos devuelva el objeto que queremos. La clase completa puede verse en el C.5 del anexo.

Tras todos estos pasos intermedios, por fin llegamos al final de nuestro ejemplo: el *bean* de CDI que usará nuestra biblioteca, tal y como vemos en código C.6 del anexo. Lo más destacable aquí quizás sea mencionar que el objeto *metadata* esté siendo inyectado y sea el método productor el que se encarga del *binding* entre el JAR externo y nuestro proyecto principal.

Tras esto, ya tenemos nuestra primera prueba realizada con éxito. Antes de realizar una valoración más detallada, hay algo que destacar: si queremos usar una biblioteca realizada en JavaEE, necesitamos que nuestro servidor corra un contenedor de EJB o CDI (como es nuestro caso).

Aunque pudiera darse el caso, es algo extraño tener una biblioteca realizada en JavaEE cuando va a importarse en proyectos que no usen esta especificación. Por tanto, descartamos realizar el ejemplo de *Biblioteca en JavaEE - Proyecto principal en Spring*.

Biblioteca en Spring - Proyecto principal en JavaEE

En esta sección seremos algo más escuetos, puesto que en la anterior hemos analizado cada detalle a fin de aclarar todos los conceptos de lo que buscamos conseguir. Ahora queremos ver las diferencias existentes en la biblioteca y cómo ésta se incorpora a un proyecto JavaEE. Nótese que seguimos teniendo la necesidad de añadir las

dependencias de Weld, ya que no hemos cambiado de servidor en ningún momento.

Antes de nada, hay que añadir las dependencias de Spring en nuestro POM, además de declarar el *packaging* del proyecto como JAR. Todo esto se puede apreciar en el C.7 del anexo.

Hay que hacer notar que la configuración en Spring es algo más compleja (o, al menos, lo es para alguien que venga directamente desde el mundo JavaEE 7). Hay tres formas de realizarla: desde un XML tradicional, con anotaciones o desde una clase anotada con *@Configuration*. En concreto, para este ejemplo simple, hemos optado por usar el clásico XML. Allí declaramos nuestro *bean* con un *scope singleton*. El *applicationContext.xml* lo podemos ver en el C.8 del anexo.

Tras esto, creamos el *SingletonMapBean* con un código muy similar al de C.5. La única diferencia es que aquí no hay ninguna anotación propia de JavaEE. Finalmente, con Maven, hacemos *install-file* y ya tendremos el JAR instalado en nuestro repositorio local.

Ahora sólo tenemos que agregarlo en nuestro proyecto JavaEE; pero este paso es trivial si tenemos la información del apartado anterior. Volvemos a crear una clase con un método productor que nos dé el objeto y, con esto, ya puede ser inyectado donde haga falta. En definitiva, repetimos los códigos descritos en C.5 y C.6. Si lo usamos como en el *servlet* creado en C.4, obtendremos una salida muy similar. En definitva, volvemos a conseguir lo que nos habíamos propuesto.

Biblioteca en Spring - Proyecto principal en Spring

Y hemos llegado al último de los ejemplos, usando Spring plenamente. Esto nos ofrece una enorme ventaja respecto a los anteriores: ya no tenemos que incorporar el contenedor de CDI (Weld) como dependencia (pero, al fin y al cabo, vamos a introducir otro motor de inyección de dependencias con Spring).

El proceso de crear la biblioteca en Spring ya acaba de ser detallado en la sección anterior, así que ahora nos centraremos en crear el proyecto Spring para usarla. Si empezamos por la dependencias necesarias, además del propio *core* de Spring vamos a necesitar MVC y algunos espacios de nombres (JSTL, Apache Taglibs). Esto es, tendremos algo similar al fragmento C.9 del anexo.

Tras esto, siguiendo los principios de la arquitectura MVC de Spring, definimos un *servlet* que escuche todas las peticiones, las redirija a un JSP cuyo controlador renderice algo similar a lo que obteníamos antes. No merece la pena comentar al detalle

todo lo necesario para montar la aplicación web con MVC, pues en la documentación de Spring encontramos bastante información. Sin embargo, en el C.10 del anexo mostramos cómo sería este controlador. Cabe destacar que, salvo las particularidades de MVC, el código es casi idéntico al que escribimos en C.6.

5.1.3 Eligiendo un *framework* para la biblioteca

Habiendo realizado este estudio detallado, Spring parece ser la solución ideal en cuanto a portabilidad de servidores, mientras que JavaEE nos aporta la estabilidad y respaldo que unos estándares sólidos nos ofrecen.

Es una decisión compleja y, con objeto de facilitar la integración con otros proyectos, la opción más indicada sería Spring. No obstante, puesto que JavaEE no se encuentra en los planes de estudio de la titulación y el autor del presente proyecto tiene experiencia laboral previa con este este *stack* tecnológico, hemos decidido realizar este *framework* para realizar la implementación.

Parte IV

Diseño

Requisitos

Failure comes only when we forget our ideals and objectives and principles.

Jawaharlal Nehru (1889–1964),
First Prime Minister of India

En este capítulo se pretende establecer los objetivos generales del proyecto a fin de poder comenzar con la arquitectura propuesta para la solución.

6.1 REQUISITOS TECNOLÓGICOS

Como ya vimos en §4.3.10, el modelo de cobro y gestión de una API es una tarea compleja y, justo por ello, existen numerosas soluciones de terceros, tanto comerciales como *Open Source*, que nos facilitan la tarea. Sin embargo, casi todas las plataformas se centran en lo mismo: limitar peticiones y establecer planes en función de las características que se ofrecen.

Si asumimos que existe una biblioteca para la gestión de acuerdos a nivel de servicio, podríamos usarla directamente y tendríamos una API con acuerdos explícitos. No obstante, el objetivo del proyecto es distinto: *se pretende crear una biblioteca que cualquier API pueda incorporar y que, con añadir un acuerdo explícito, sea suficiente para obtener una API que gestione las limitaciones de peticiones basado en SLA*.

6.1.1 Caso de estudio

Para motivar nuestro proyecto en un caso real, en la Figura §6.1 encontramos un plan de precios de una API comercial.

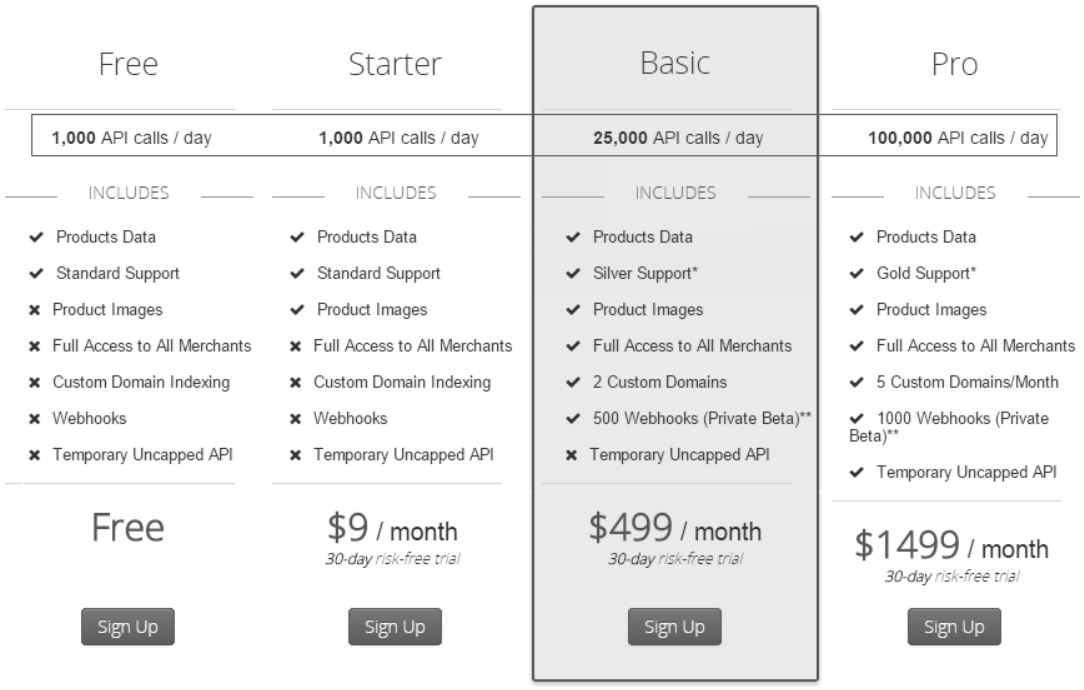

Figura 6.1: *Pricing plan* de Semantics3.

En concreto, en la Figura §6.1 se muestra el plan de precios de la empresa *Seman-*

tics3, una compañía que ofrece una API de datos de productos de comercio electrónico, códigos de barra, precios en diferentes tiendas, etc. Se pueden apreciar cuatro planes diferentes basados, fundamentalmente, en el número de peticiones diarias. Asimismo se ofrecen diferentes niveles de soporte y características para cada plan, por ejemplo, en el *plan gratuito* no tenemos acceso completo a la API ni podemos indexar dominios personalizados. Aunque detallaremos más tarde una guía de uso de nuestra solución, resulta evidente que para tener un API Gateway que gobierne esta API, es preciso disponer de plantillas de acuerdo definidas, en las que se establezcan los límites de 1000, 25000 y 100000 peticiones diarias. En una posible ampliación de la solución se podrían considerar otras propiedades personalizables distintas al número de peticiones.

Inspirados en este caso real, para este caso de estudio vamos a describir una API sintética sencilla en la que es necesario gestionar únicamente el número de peticiones. La llamaremos *PapamoscasAPI* y mostrará datos de avistamientos sobre aves. En la Figura §6.2 vemos el *pricing plan* asociado.

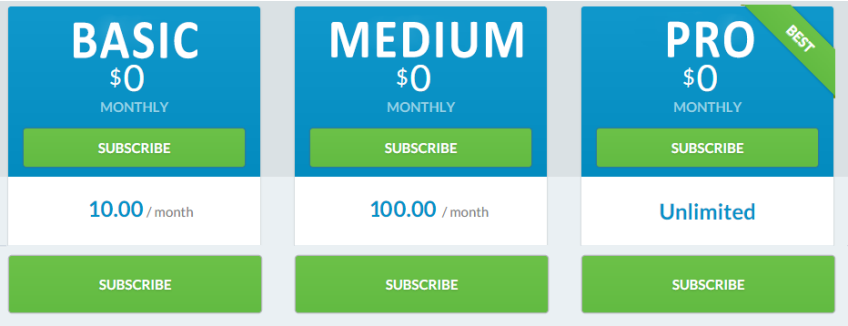

Figura 6.2: Plan de precios de *PapamoscasAPI*

Concretamente es posible ver que existen tres planes claramente diferenciados, de 10, 100 e infinitas peticiones respectivamente. Sin embargo, a pesar de ser una API completamente inventada, no difiere demasiado con la API de la Figura §6.1.[1]

[1]En la sección §10 se puede encontrar la resolución de este caso de estudio.

Arquitectura

A lot of people assume that creating software is purely a solitary activity where you sit in an office with the door closed all day and write lots of code.

Bill Gates (1955),
Microsoft co-founder

Durante el presente capítulo se introducen los aspectos lógicos de la solución propuesta, dando los detalles necesarios para su correcta comprensión, pero sin entrar en el detalle de la implementación. Por otra parte, se hace mención a la biblioteca de gestión de acuerdos que subyace sobre la solución.

7.1 ARQUITECTURA DE LA SOLUCIÓN

Como se ha mencionado anteriormente, el objetivo del proyecto es crear una biblioteca que cualquier API pueda incorporar para que, al añadir un acuerdo explícito, sea suficiente para convertir a tal API en una que gestione las limitaciones de peticiones basado en SLAs explícitos.

Actualmente las APIs deben implementar toda la lógica de la autorización de peticiones manualmente, por lo que un cambio en políticas en la empresa requerirá realizar un cambio en la lógica de la aplicación. En un simple diagrama de contexto, podemos representar esta situación tal y como aparece en la Figura §7.1.

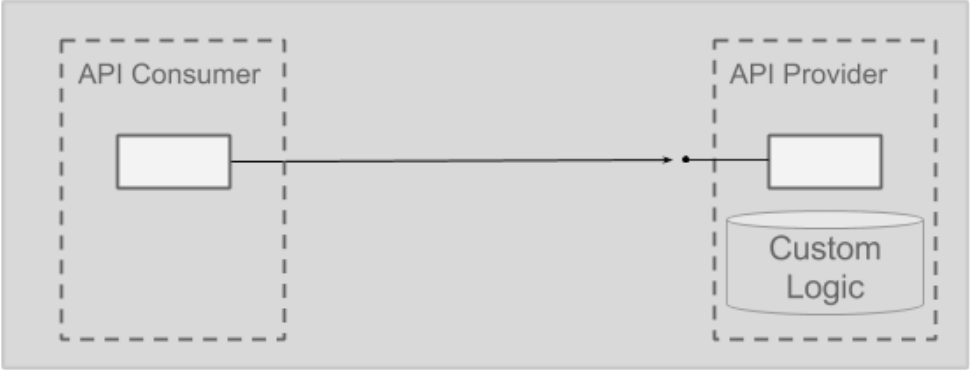

Figura 7.1: Situación de las APIs actuales.

Con la solución que se propone, se incorpora la figura de un *API Gateway* que es el responsable de la autorización de peticiones basándose en una biblioteca de gestión de SLAs que incorpora. Para facilitar la comprensión de la arquitectura lógica que proponemos, es preciso realizar un sencillo diagrama de contexto en el que se detalle el papel que juega esta biblioteca en la relación entre proveedor y consumidor del servicio; en la Figura §A.1 se muestra tal diagrama.

El consumidor realiza peticiones al *endpoint* de la API, pero el *API Gateway* debe decidir previamente si la petición del usuario que la realiza es autorizada o no. De ser así, actualiza el estado del acuerdo del usuario y sirve la petición (a través de la API).

La propuesta que se presenta en este proyecto consta de dos partes diferenciadas; la principal, la biblioteca de gestión de *princing* basada en SLAs y, a modo de ejemplo, una API REST muy simple, que sirve de prueba de concepto de lo que queremos mostrar.

Como se mencionó en capítulos anteriores, la arquitectura tecnológica a usar es *Java Enterprise Edition*, más conocida como JavaEE. Además, la API estará realizada

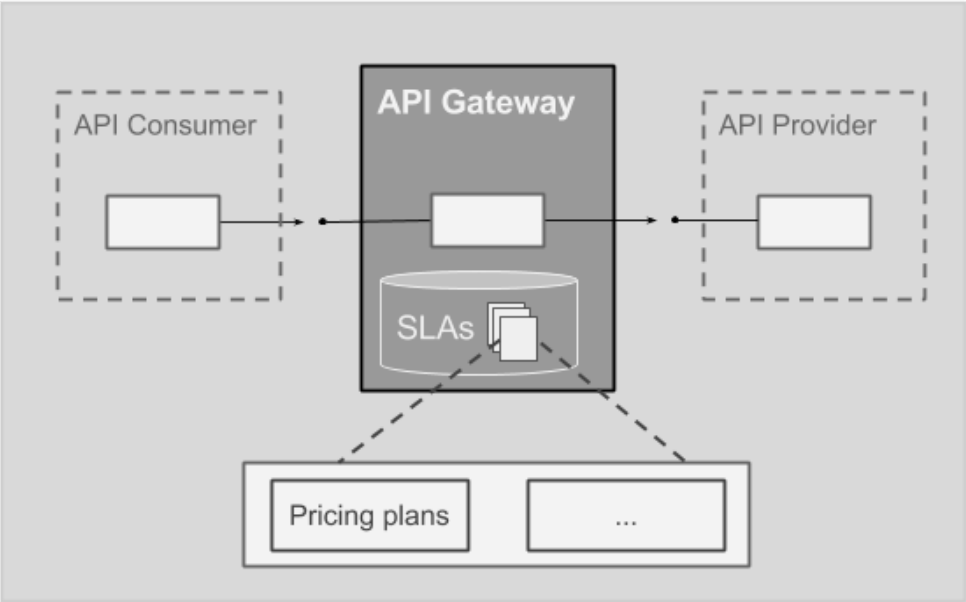

Figura 7.2: Situación de las APIs tras la solución propuesta.

siguiendo las especificaciones de JAX-RS, cosa que nos facilitará la tarea de anotar los diversos métodos de la API.

Profundizado algo más en la estructura de la biblioteca que estamos desarrollando, partimos de una *Agreement Management Library* (AML), que será la base de nuestro proyecto[1].

Sobre esta biblioteca se encuentra el filtro de peticiones. Un filtro (o cadena de ellos), en el contexto de *Java Servlet*, se corresponde con aquella clase que es llamada antes de que la petición sea resuelta. Se puede entender como una forma de interceptación de peticiones y nos puede recordar a la orientación a aspectos. Este filtro será el que realice la lógica y la llamada a la biblioteca.

Finalmente, para simplificar el uso de esta biblioteca de apoyo al *pricing* en APIs REST, hemos hecho uso de las anotaciones disponibles en Java. De esta forma, anotando la API objetivo y cargando el SLA correspondiente, tendremos ya resuelto el problema de limitar peticiones en función del plan suscrito por el usuario.

En la Figura §7.3 se puede observar una representación de esta estructura.

En concreto, tenemos como base la biblioteca AML, la cual importamos en esta so-

[1]Como se ha comentado previamente, AML surge como fruto del trabajo de algunos integrantes del grupo de investigación de Ingeniería del Software Aplicada. Si bien durante este proyecto se han realizado colaboraciones puntuales, queda fuera del ámbito del presente trabajo de fin de grado.

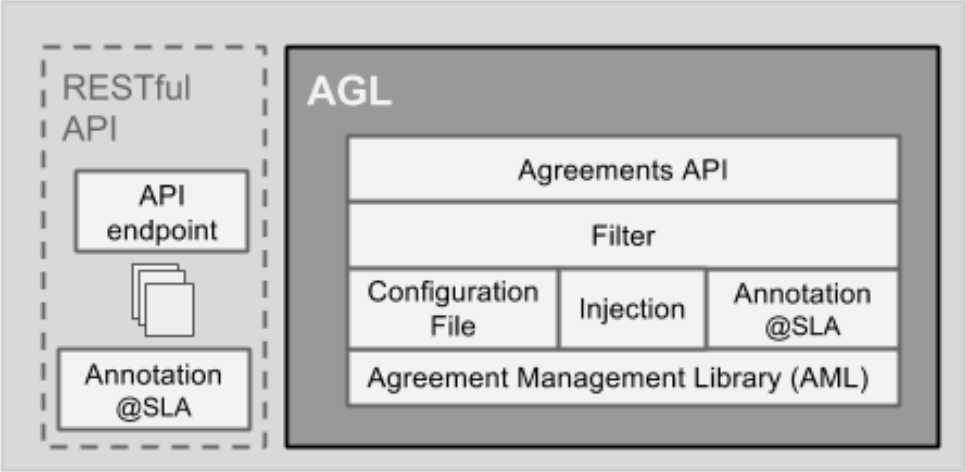

Figura 7.3: Esquema detallado de la solución propuesta.

lución (AGL) que proponemos. Por otra parte, el uso de la anotación @SLA nos permite simplificar el uso de AGL; además, al incorporar un archivo de configuración externo, no acoplamos la nomenclatura del acuerdo con la de la lógica de la biblioteca. Por otra parte tenemos el filtro, que será el motor de procesamiento de peticiones para verificar si son aceptadas o no. Finalmente, se incorpora una simple API que nos muestra el estado actual del acuerdo. No obstante, en secciones posteriores hablaremos más profundamente sobre esto.

7.2 Arquitectura de AML

Hasta ahora hemos hablado de una *biblioteca de gestión de acuerdos*, que si bien ha sido desarrollada de forma externa a este proyecto, ha sido necesario comprender su uso para poder integrarla adecuadamente.

La así denominada AML (*Agreement Management Library*), en su versión actual, ofrece numerosas funcionalidades: procesado de SLA especificado en *WS-Agreement* o *iAgree* [9], almacenamiento en memoria del acuerdo y conversión a un modelo de datos común para acuerdos. Una vez que el modelo ha sido definido, podemos realizar diferentes operaciones con los diferentes acuerdos cargados. Todas ellas han sido fruto de la línea investigadora del grupo y han sido implementadas siguiendo las tesis doctorales en las que se describen (por ejemplo, [8]).

Puesto que no usaremos ninguna funcionalidad de análisis, detallaremos aquí el proceso necesario desde que un acuerdo es definido y formalizado hasta que dispone-

mos del modelo del acuerdo y podemos evaluar expresiones sobre el mismo.

A nivel conceptual y de forma simplificada, los elementos que conforman la *Agreement Management Library* son los que observamos en la Figura §7.4.

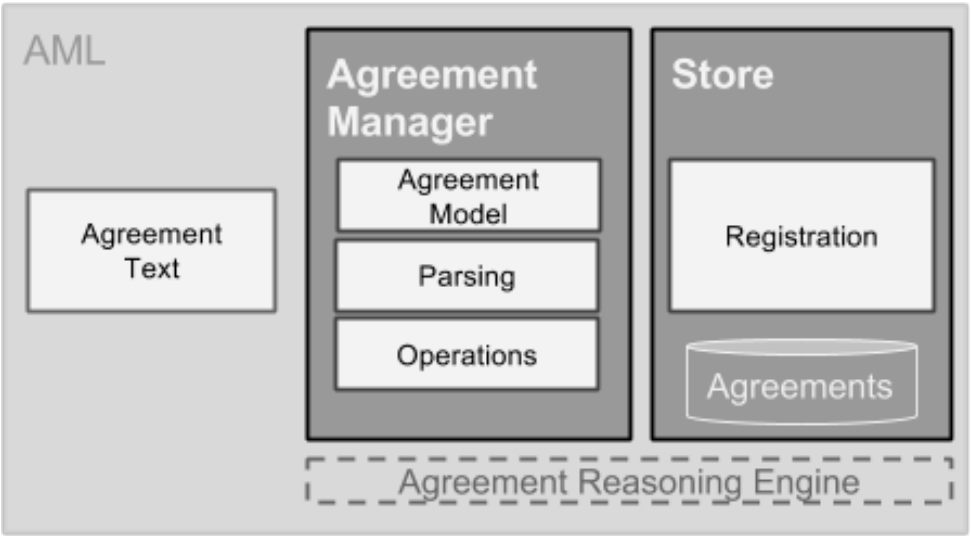

Figura 7.4: Diagrama simplificado de AML.

7.2.1 WS-Agreement y acuerdos

Antes de nada, conviene mencionar la estructura de un documento de acuerdo formalizado en *WS-Agreement* a través de la Figura §7.5 perteneciente a la Sección 2.2 de [9].

Hablar con detalle de la formalización de un documento de acuerdo y la negociación entre plantilla, oferta y acuerdo definitivo sería una tarea que se escapa de los objetivos del presente trabajo de fin de grado y además ha sido un tema ampliamente tratado en numerosas publicaciones del grupo de investigación. No obstante, es preciso dar nociones básicas que permitan comprender, a grandes rasgos, las características de un acuerdo a nivel de servicio.

Un SLA definido en *WS-Agreement* debe contener un identificador de acuerdo, un contexto de acuerdo y unos términos de acuerdo:

1. El **contexto** tiene información general sobre el propio acuerdo, el tiempo de vida, métricas y tipos de datos del dominio de éstas.

2. Los **términos** clasifican en dos grupos: los términos del servicio y los términos de

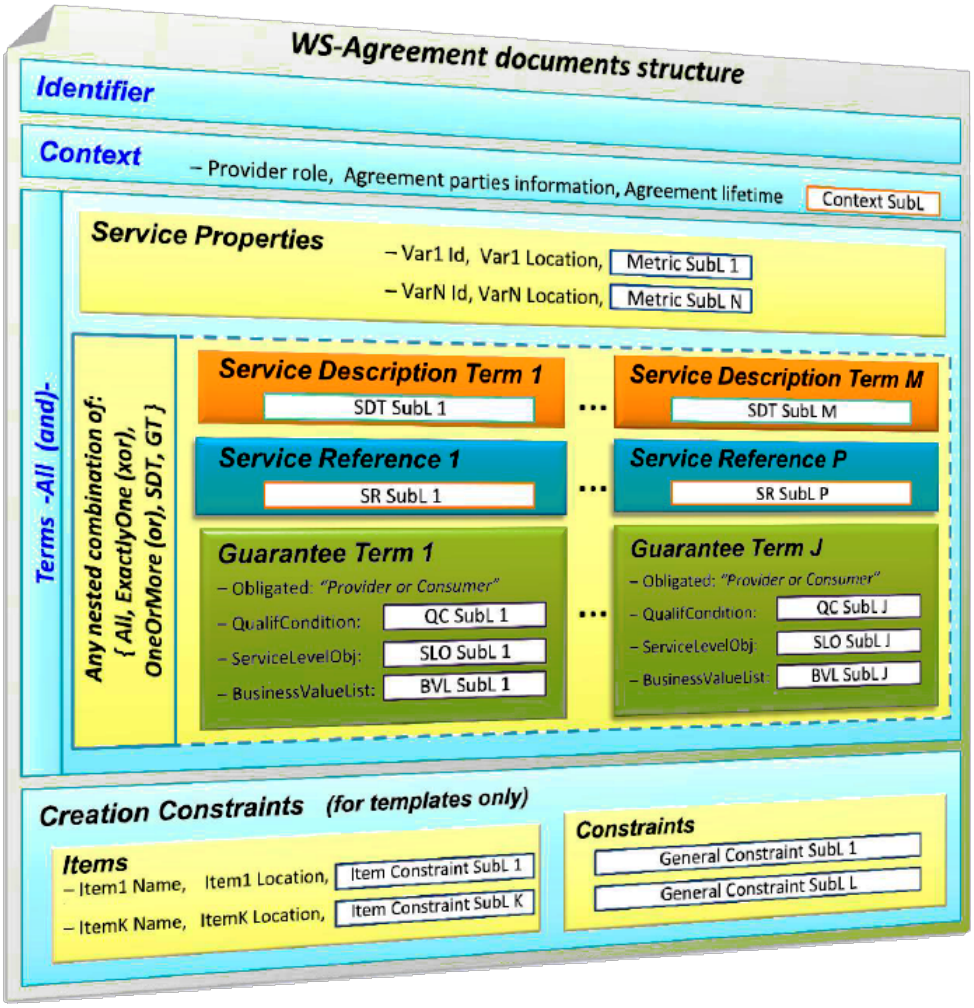

Figura 7.5: Estructura de un documento *WS-Agreement*.

garantía:

a) Los **términos del servicio** definen las características del servicio y se diferencian según su objetivo en:

1) **Referencia del servicio**: *endpoint* de la interfaz del servicio.
2) **Descripción del servicio**: características del servicio acordadas entre consumidor y proveedor.
3) **Propiedades del servicio**: especifican las variables monitorizables que serán usadas para definir los términos de garantía. Estas propiedades deben estar especificadas en las métricas definidas en el contexto.

b) Los **términos de garantía** (Guarantee Term GT) especifican las diferentes expresiones (*Service Level Objective* SLO) que el respondedor (usualmente, el proveedor del servicio) debe cumplir. Además, estos términos pueden tener

una condición de aplicación (*Qualifying Condition*, QT) que determina si esa expresión debe ser usada o no. Finalmente, se pueden añadir compensaciones, esto es, recompensas o penalizaciones. De esta forma, un GT con compensaciones determinaría si el consumidor o proveedor puede solicitar una recompensa o debe ser penalizado. Mediante el diagrama de clases adaptado de [9] de la Figura §7.6 representamos las distintas partes que conforman un término de garantía. Nótese que la coloración gris indica que actualmente AML no ofrece soporte a compensaciones en acuerdos [14].

En definitiva, estos términos de garantía nos servirán para modelar las restricciones sobre las peticiones dentro de los planes de precio de una API. Definiendo una variable (*property*) donde actualicemos el valor del número de peticiones del usuario en concreto, basta con evaluar el término de garantía correspondiente para determinar si una petición es aceptada o no. En el fragmento 7.1 vemos un ejemplo.

```
1  Template IDENTIFICADOR version 1.0
2    Initiator: "NOMBRE DEL INICIADOR DEL SERVICIO";
3    Provider "NOMBRE DEL PROVEEDOR" as Responder;
4    Metrics:
5      int: integer[0..10000]; //rango a elegir
6  AgreementTerms
7    Service NOMBRE DEL SERVICIO availableAt "ENDPOINT DEL SERVICIO"
8      GlobalDescription
9    MonitorableProperties
10     global:
11       REQUEST_VAR: int = 0;
12   GuaranteeTerms
13     GT_NAME: Consumer guarantees REQUEST_VAR <=MAX_REQUESTS; //
     ↪ MAX_REQUESTS es fijo
14 EndTemplate
```

Código 7.1: Estructura de acuerdo.

En este caso, el término de garantía GT_NAME evalúa que REQUEST_VAR <= MAX_REQUESTS. Estos nombres pueden ser escogidos libremente, pero han de ser indicados de alguna forma a la hora de usar nuestra propuesta. Este tema será tratado en la siguiente sección.

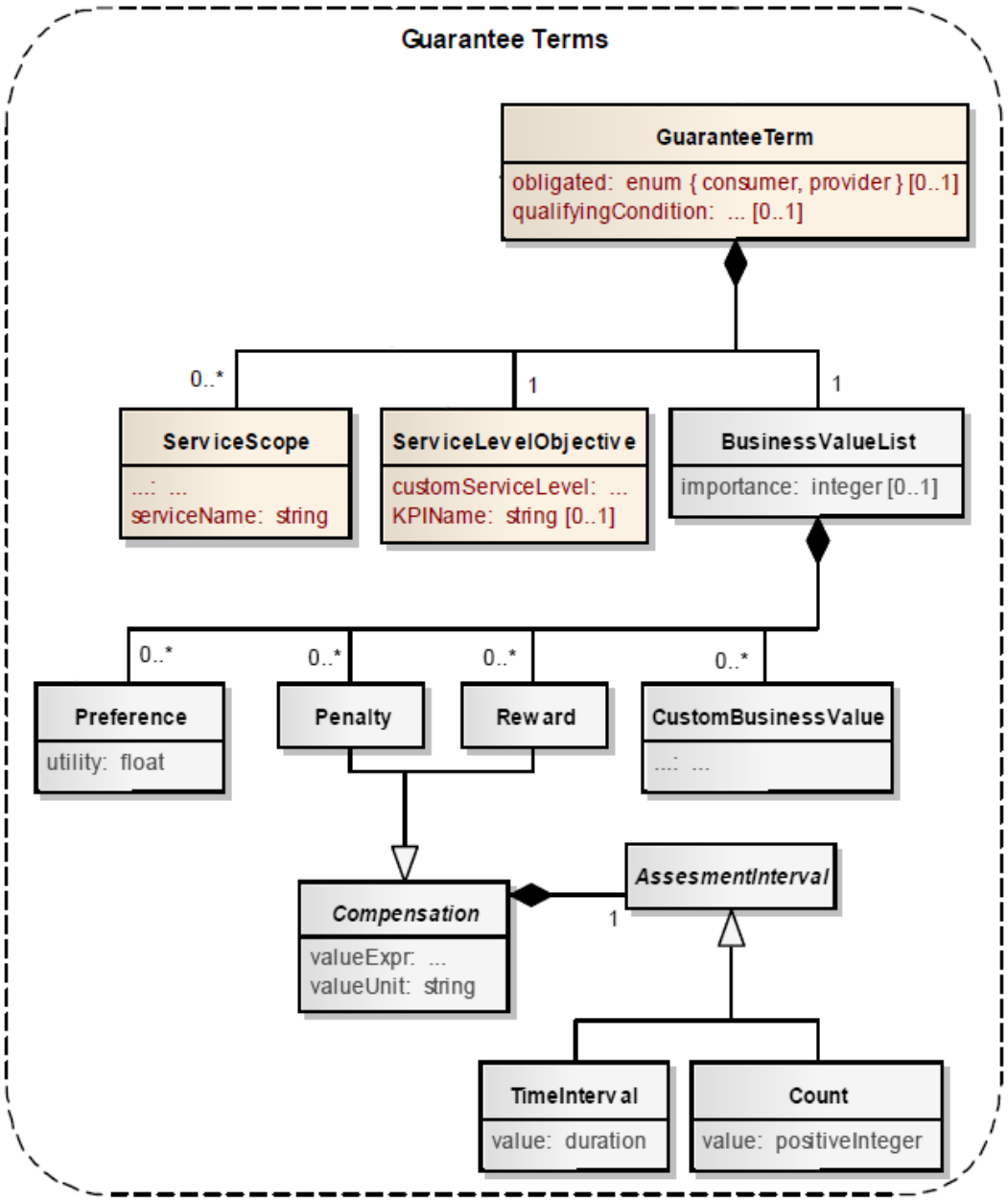

Figura 7.6: Diagrama de clases de los términos de garantía de un acuerdo.

Parte V

Implementación

Implementación de AGL

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Melvyn Conway (1967),
Computer Scientist

Este es el capítulo central del presente trabajo de fin de grado, donde se realiza el desarrollo de la solución anteriormente propuesta. En primer lugar se realiza un resumen de las tecnologías sobre las que se realiza el proyecto. Posteriormente se entra en los detalles de la implementación, desde la importación de AML, la creación de la anotación, el desarrollo del filtro y la generación de una API REST que muestre el estado del acuerdo en cada instante.

8.1 Stack tecnológico

Entramos ya en el núcleo de este proyecto, la implementación de la solución propuesta en este trabajo de fin de grado. Conviene antes destacar las tecnologías que van a ser usadas. Además, en la Figura §8.1 se aprecia un diagrama que condensa esta información.

- **Lenguaje**: Java.
- **Framework**: Java EE 7.
- **Elementos destacables**: Servlet 3.1, Annotations 1.2, JPA 2.1, EJB 3.2, CDI 1.1, MySQL 5.6
- **Especificación de servicios web**: JAX-RS (2.0), con implementación de Jersey.
- **Servidor de aplicaciones**: Glassfish 4.1.

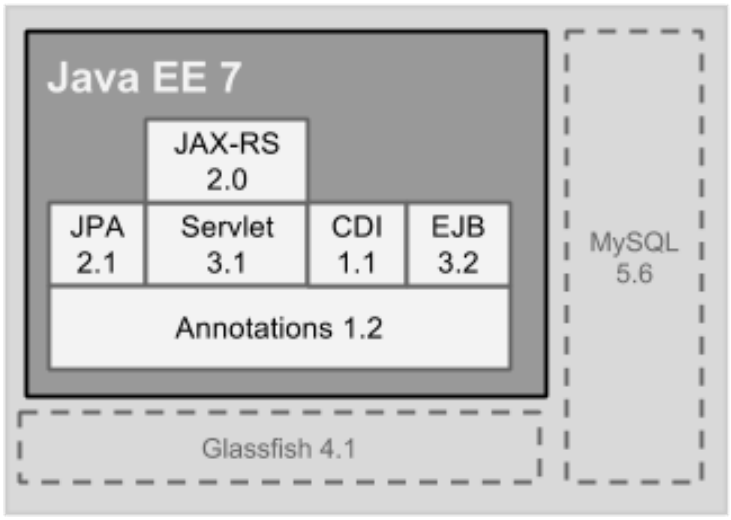

Figura 8.1: Representación del *stack* tecnológico empleado.

8.2 DESARROLLO

8.2.1 Importación de AML

El desarrollo de este proyecto es la parte principal que nos ocupa, pues tras realizar todo lo que aquí se detallará dispondremos de un JAR que cualquier proyecto Java podrá importar para hacer uso de los servicios de gestión de peticiones basada en acuerdos explícitos.

En primer lugar es necesario importar la biblioteca AML de forma similar a como hicimos en §5. La única diferencia es que la gestión de dependencias se realiza directamente con Maven, pues el grupo tiene un repositorio *Nexus* de artefactos JAR desplegado en la red. En el fragmento 8.1 se encuentra lo que debemos añadir al *pom.xml*, además de las dependencias necesarias de la API de JavaEE.

```
1  <repositories>
2    <repository>
3      <id>AML-releases</id>
4      <name>ISA</name>
5      <url>http:// clinker.isagroup.es/ nexus/ content/ repositories/
       ↪ AML</url>
6    </repository>
7  </repositories>
8
9  <dependencies>
10   <dependency>
11     <groupId>es.us.isa</groupId>
12     <artifactId>AML</artifactId>
13     <version>0.1.2</version>
14   </dependency>
15 </dependencies>
```

Código 8.1: Extracto del POM del proyecto.

Una vez que hemos importado AML, es necesario instanciar la biblioteca. Para ello, y aprovechando las ventajas de JavaEE, definimos un *bean* con *scope* de aplicación (en definitiva, no es más que una implementación del patrón *singleton* dentro del contexto de CDI) que instancie AML y tenga un método que *produzca* tal objeto. Nótese que con este diseño, nos desacoplamos de la versión de la biblioteca; al estar en un contexto

de inyección de dependencias e inversión del control, basta con anotar un método con *@Produces* y en otro sitio inyectar un objeto del mismo tipo. En el fragmento 8.2 se puede apreciar, el código completo se encuentra en el C.20 del Anexo.

```
1  @ApplicationScoped
2  public class AMLHandler{
3    @PostConstruct
4    public void init() {
5      am = new AgreementManager();
6    }
7    @Produces
8    public AgreementManager getAML() {
9      return am;
10   }
11 }
```

Código 8.2: Fragmento de la clase AgreementsServlet.

Como se puede ver en las líneas 3-6, basta con realizar la instanciación en el método anotado como *@PostConstruct* y luego devolverlo en el *getter*, líneas 7-10.

8.2.2 Creación de @SLA

Una vez realizado esto podemos hacer uso de todas las funcionalidades que AML ofrece. Sin embargo, puesto que queremos ofrecer una forma sencilla para que una API REST la use, vamos a introducir una anotación propia que debemos crear. Esta tarea es sencilla, tal y como se observa en el fragmento 8.3. Nótese que las líneas 1-4 son otras anotaciones que describiremos y la 5 es la propia declaración de la interfaz.

```
1 @SLA
2 @Retention(RetentionPolicy.RUNTIME)
3 @Target({ElementType.TYPE, ElementType.METHOD})
4 @NameBinding
5 public @interface SLA {
6 }
```

Código 8.3: Anotación SLA.

En primer lugar, declaramos la *@interface* SLA sin ningún método. Además dicha interfaz la anotamos con:

- **@SLA**: nombre con el que va a ser llamada tal anotación.
- **@Retention(RetentionPolicy.RUNTIME)**: la anotación es tanto en tiempo de compilación, aunque la máquina virtual la retiene en *runtime* a fin de poder llamada por reflexión.
- **@Target(ElementType.TYPE, ElementType.METHOD)**: indica a qué elementos le son aplicables esta anotación. En nuestro caso, a los *tipos* y a los *métodos*.
- **@NameBinding**: es una meta-anotación que indica a JAX-RS que aquella que estamos defiendo es aplicable a filtros e interceptores.

En este punto de la implementación probablemente no se comprenda a la perfección la utilidad de definir esta anotación, sin embargo, al añadir el concepto de filtro, veremos que todo lo anterior encaja con nuestro propósito.

8.2.3 Implementación del filtro

El siguiente elemento a desarrollar es el filtro de peticiones, para ello, extendemos de la clase del filtro abstracto, que, al usar JAX-RS y su implementación *Jersey*, se denomina CONTAINERREQUESTFILTER. Si bien el código completo se puede encontrar en el C.21 del Anexo, a continuación veremos y explicaremos algunos fragmentos de esa clase.

En primer lugar, tal y como se aprecia en 8.4, de todas las anotaciones que tienen un *binding* con el filtro que estamos definiendo (línea 4), nos quedamos con *SLA*. Esto se realiza a través de reflexión (línea 3), de ahí la necesidad de una *retention policy* fijada en *runtime*.

```
1 public void filter(ContainerRequestContext requestContext) throws
    ↪ IOException {
2 Boolean isSLA = false;
3 for (Annotation annotation : resourceInfo.
    ↪ getResourceMethod().getDeclaredAnnotations()) {
4         if (annotation instanceof SLA) {
5                 isSLA = true;
6                 break;
7         }
8 }
```

Código 8.4: Fragmento 1 de la clase SLARequestFilter.

A continuación, como se ve en 8.5, cogemos el *token* del usuario que, según definimos, nos llega a través de un *query parameter* (líneas 5-6), es decir, algo de la forma ?USER=TOKEN. Esto podría ser cambiado por un *header* HTTP simplemente cambiando el método llamado desde el contexto de la petición.

Finalmente, aumentamos el valor de las peticiones realizadas (línea 7) y seguidamente vemos si la petición debe ser o no servida (llínea 8). En el caso de que se permita, no se hace nada más; en otro caso, el filtro interrumpe el flujo normal lanzando una *WebApplicationException* con el código de estado HTTP correspondiente.

```
1  if (isSLA) {
2          //SLA aware method
3          List<String> token = requestContext.getUriInfo().
               ↪ getQueryParameters().
               ↪ get(config.getProperty("queryParamToken"));
4          String clientId;
5          if (token != null && !token.isEmpty()) {
6                  clientId = token.get(0);
7                  onRequestDone(clientId); //increasing request number
8                  if (!isRequestAuthorized(clientId)) {
9                          throw new WebApplicationException(
                               ↪ Response.Status.PAYMENT_REQUIRED);
10                 }
11         } else {
12                 throw new WebApplicationException(
                       ↪ Response.Status.BAD_REQUEST);
13         }
14 }
```

Código 8.5: Fragmento 2 de la clase SLARequestFilter.

Hasta ahora no hemos hecho uso de la biblioteca AML, esto es, no hemos añadido nada nuevo a lo que un filtro cualquiera ya haga. La solución propuesta por este trabajo pretende asociar conceptos propios de los acuerdos a nivel de servicio con el detalle de la implementación. En particular es preciso detallar los métodos ISREQUESTAUTHORIZED(CLIENTID) y ONREQUESTDONE(CLIENTID).

Debemos inyectar la biblioteca AML, en concreto, la clase encargada de almacenar y registrar acuerdos, llamada *Store*[1]. Una vez hecho esto, podemos implementar tales

[1] Aunque no se ha mostrado explícitamente en el Fragmento C.20, existe un *getter* para el objeto *Store*.

métodos, como se describe en el fragmento 8.6.

```
1  private boolean isRequestAuthorized(String clientId) {
2      Agreement agreement = store.getAgreement(clientId);
3      return agreement.evaluateGT( config.getProperty("requestGT"));
4    }
5
6  private void onRequestDone(String clientId) {
7    Integer numReq;
8    Agreement agreement = store.getAgreement(clientId);
9          numReq = agreement.getProperty(
               ↪ config.getProperty("requestVar")).intValue();
10         numReq++;
11         agreement.setProperty( config.getProperty("requestVar"),
               ↪ numReq);
12
13 }
```

Código 8.6: Fragmento de la clase SLARequestFilter.

Para comprender lo que se muestra en el fragmento es preciso conocer lo que se detalla en §7.2.1, pues AML usa un modelo de datos de acuerdo muy similar al que se comentaba previamente.

En primer lugar, el ISREQUESTAUTHORIZED nos determina si una petición debe o no ser aceptada. Para ello, coge el acuerdo del usuario que ha sido previamente registrado (en la siguiente sección trataremos el registro de acuerdos) y llama al método EVALUATEGT, pasándole el nombre del término de garantía.

Nótese que tal nombre se obtiene a partir de la evaluación de cierto objeto *config* (previamente inyectado), esto es porque hemos externalizado las cadenas usadas en el acuerdo a un fichero de configuración de la clase *Properties* de Java.

Por otro lado, el método ONREQUESTDONE(CLIENTID) accede al modelo de una forma similar al anterior: obtenemos el acuerdo, de él consultamos el valor de la propiedad *requestVar*, lo incrementamos en una unidad y, finalmente, actualizamos la propiedad con el nuevo valor.

Tras hacer esto, hemos obtenido un filtro que toma los valores a evaluar a partir de un acuerdo; algo completamente desacoplado de la lógica de la API y fácilmente integrable en en cualquier desarrollo existente.

Llegados a este punto, sería conveniente conocer el estado del acuerdo a lo largo del tiempo, para así poder tomar decisiones en función del mismo. Por ejemplo, un agente que lo monitorizara podría establecer compensaciones en función del cumplimiento del acuerdo. Sobre esto existe abundante documentación, por ejemplo [16].

8.2.4 API de Agreements

Por otro lado, no existe ningún estándar para la representación RESTful del estado del acuerdo (aunque existen propuestas de algunos autores como [6]), cosa que nos lleva a optar por una opción sencilla y al alcance de un trabajo de fin de grado.

Vamos a diseñar una API REST que nos muestre, en formato textual plano dos tipos de recursos; por un lado, todos los acuerdos de los usuarios del sistema y, por otro, el acuerdo de un determinado usuario. Para ello, haremos uso de la especificación JAX-RS para definir una API REST sencilla.

En primer lugar definimos un *bean* sin estado en el contexto de *EJB* y lo anotamos con *@Path(«agreements»)* para indicar el *endpoint* del recurso. Posteriormente declaramos un método que produce una respuesta en formato *text/plain*. En el fragmento 8.7 se observa lo más relevante de la clase.

```
1  @Stateless
2  @Path("agreements")
3  public class AgreementsREST {
4    @GET
5    @Path("{clientId}")
6    @Produces("text/plain")
7    public Response find(@PathParam("clientId") String clientId) {
8      if (amlHandler.getAML().getStoreManager().
     ↪ getAgreementMap().containsKey(clientId)) {
9        return Response.ok(getTranslator().
     ↪ export(amlHandler.getAML().getStoreManager().
     ↪ getAgreementMap().get(clientId))). build();
10     } else {
11       return Response.status(Response.Status. NO_CONTENT).build();
12     }
13   }
14 }
```

Código 8.7: Clase AgreementsRESTFragment.

No es complejo comprender la sintaxis de la definición de servicios RESTful mediante JAX-RS. Hemos anotado el método *find* como *@GET* y recibe el identificador del cliente en la propia URI. A continuación, desde la AML cogemos el mapa de acuerdos y obtenemos aquel que coincide con el identificador; si no existe, se envía el código de estado correspondiente.

Con este ejemplo es suficiente para ver cómo se implementa la API, no obstante, hace falta otra clase más hacer funcionar los servicios con JAX-RS. En el C.22 del Anexo se encuentra esta clase y en C.23, la clase anterior completa.

Validación

To infinity and beyond.

Buzz Lightyear (1995),
Toy Story

Durante este capítulo se analizarán las soluciones propuestas anteriormente para que, dada una API inicial, se realice la autorización de peticiones basada en una cuota determinada. Estas soluciones son: sin usar ninguna biblioteca de gestión de SLAs, usando AML y, finalmente, empleando AGL.

Tras haber mostrado la solución propuesta es preciso analizar si es realmente necesario el uso de una biblioteca que nos facilite la gestión de la autorización de peticiones en función de un acuerdo explícito. Por un lado, es posible realizar otros diseños que no requieran ningún concepto de SLA y por otro, podemos usar AML directamente, sin depender de JavaEE ni añadir más dependencias al proyecto base.

Por lo anterior, es importante la existencia de este capítulo, donde se explicita cómo, dada una API inicial, realizar la autorización de peticiones basada en una cuota determinada de tres formas distintas: sin usar ninguna biblioteca de gestión de SLAs, usando AML y, finalmente, empleando AGL.

9.1 Implementación base

En el caso de no emplear ninguna biblioteca, debemos guardar de alguna forma la relación entre tokens y número de peticiones disponibles. En el caso más sencillo, asumimos que se trata de un simple mapa en memoria donde a cada cadena de texto se le asigna un determinado entero. En un caso real usaríamos JPA o algún otro tipo de persistencia.

El mapa se crea sobre un *singleton* que realiza la inicialización de la instancia del handler y crea una entrada en el mapa por cada uno de los usuarios de la API. Además, ofrece una pequeña fachada del mapa creado para hacer más sencilla la lógica del filtro. El código completo puede verse en el C.12 del Anexo.

La lógica sobre si una petición debe o no ser aceptada es extremadamente simple, tal y como se puede apreciar en 9.1.

```
1 private boolean authorizeRequest(HttpServletRequest req) {
2         Integer availableRequests = Helper_noAML.getInstance().
             ↪ getAvailableRequestsFromUser(clientId);
3         return availableRequests > 0 || availableRequests == -1;
4 }
```

Código 9.1: Método authorizeRequest sin biblioteca.

Dada una petición, hemos extraído el token del usuario y con eso podemos obtener peticiones restantes del mapa en memoria. Si tiene al menos una petición disponible (o es -1, para contemplar planes ilimitados).

Si la petición fue aceptada, es preciso decrementar el número de peticiones dispo-

nibles (o dejarlo como está en planes ilimitados). Esto puede verse en 9.2.

```
1 private void requestDone(HttpServletRequest req) {
2         Integer availableRequests = Helper_noAML.getInstance().
3         getAvailableRequestsFromUser(clientId);
4         if (availableRequests > 0) {
5                 availableRequests--;
6                 Helper_noAML.getInstance().
7                 setRequests(clientId, availableRequests);
8         }
9 }
```

Código 9.2: Método requestDone sin biblioteca.

Una gran ventaja de la arquitectura que hemos usado es que permitimos que el *servlet* sea ajeno a toda la lógica de autorizaciones, pues usamos un filtro (o cadena de filtros, en el caso más general), para realizar estas tareas.

9.2 IMPLEMENTACIÓN BASADA EN AML

Para trabajar con una biblioteca de gestión de acuerdos, como AML, es preciso disponer de un acuerdo formalizado en alguna especificación, como iAgree. La implementación no es demasiado compleja una vez que se conoce el modelo de datos del acuerdo. Es importante que la biblioteca nos ofrezca operaciones de alto nivel para simplificar el acceso al modelo.

Siguiendo la arquitectura descrita en la introducción, en primer lugar debemos inicializar el sistema cargando las plantillas y registrando los usuarios de prueba con sus correspondientes acuerdos. Esta tarea la realizará una clase Helper *singleton* que tendrá también la instanciación de la biblioteca de gestión de acuerdos.

Si bien la clase completa se puede encontrar en C.17 del Anexo, la parte fundamental se aprecia a continuación en 9.3.

```
1 private static void init() {
2         agr = new AgreementManager();
3         agr.getStoreManager().registerFromFolder (new
            ↪ File("files").getAbsolutePath(), false);
4 }
5
```

```
6  private static void loadTestAgreements() {
7          loadTestAgreement("basicPlanT", "basicUser1");
8          loadTestAgreement("basicPlanT", "basicUser2");
9          loadTestAgreement("mediumPlanT", "mediumUser1");
10         loadTestAgreement("mediumPlanT", "mediumUser2");
11         loadTestAgreement("proPlanT", "proUser1");
12         loadTestAgreement("proPlanT", "proUser2");
13 }
14
15 private static void loadTestAgreement(String templateName, String
       ↪ clientId) {
16         agr.getStoreManager(). getAgreementTemplate(templateName).
              ↪ generateAgreementOffer(clientId).
              ↪ generateAgreement(clientId). register(clientId);
17 }
```

Código 9.3: Inicialización de la clase Helper con AML

Lo que aquí se hace es sencillo: se instancia la biblioteca con la configuración por defecto, aunque se le podría pasar un JSON para configurarla de forma determinada, la instanciamos, cargamos las plantillas desde fichero y las registramos en la biblioteca. Posteriormente creamos el acuerdo a partir de la plantilla (en este caso basta con clonarla) y lo registramos.

Con lo anterior ya tenemos el sistema en operación. Estudiemos ahora qué ocurre si llega una petición: el método *authorizeRequest* del filtro es el responsable de responder, por lo que deberá cargar el acuerdo correspondiente al cliente que ha efectuado la petición y comprobar el número de peticiones realizadas a través de la *Monitorable Property* correspondiente. Con esto, basta con evaluar el *Guarantee Term* correspondiente Si bien el filtro completo se encuentra en C.18, la parte fundamental la podemos ver en 9.4.

```
1  private boolean authorizeRequest(HttpServletRequest req) {
2          Agreement agreement =
              ↪ agr.getStoreManager().getAgreement(clientId);
3          if (agreement != null) {
4                  return agreement.evaluateGT("RequestTerm");
5          }
6          return false;
7  }
```

Código 9.4: Método authorizeRequest con biblioteca.

Tras esto ya sabemos si una petición debe o no ser aceptada pero aún falta el último paso: actualizar incrementar el valor de las peticiones realizadas. En este ejemplo, es bastante sencillo, pues basta con acceder a la *Monitorable Property*, incrementarla y actualizar su valor en el modelo del acuerdo. Esto lo vemos en 9.5.

```
1  private void requestDone() {
2          Integer numReq;
3          Agreement agreement = agr.getStoreManager().
               ↪ getAgreement(clientId);
4          if (agreement != null) {
5                  if (agreement.getProperty("Requests").
                       ↪ getExpression() == null) {
6                          agreement.setProperty("Requests", 0);
7                  }
8                  numReq = agreement.
                       ↪ getProperty("Requests").intValue();
9                  numReq++;
10                 agreement.setProperty("Requests", numReq);
11         }
12 }
```

Código 9.5: Método requestDone con biblioteca.

No obstante, en un caso más real, podríamos invocar a este método sólo en el caso de las peticiones exitosas o tener varios contadores y decidir en función de ellos.

9.3 Implementación basada es AGL

Al igual que en la sección anterior y puesto que seguimos trabajando con una biblioteca de gestión de acuerdos, es preciso disponer de acuerdos formalizados. En este caso además, imponemos que la API a la que le vamos a añadir el control de peticiones está realizada mediante JAX-RS. En §8.2.4 ya vimos cómo se ponía en marcha una API diseñada siguiendo esta especificación.

Asumimos que esta API o proyecto base se encuentra en un proyecto Maven. A continuación añadimos la dependencia del artefacto generado por la biblioteca y ya

podemos hacer uso de las clases que hemos desarrollado en este proyecto. Esto se aprecia en el fragmento 9.6.

```
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>tfg-api-gateway</artifactId>
  <version>${project.version}</version>
</dependency>
```

Código 9.6: Extracto del POM del proyecto base.

Con esto, ya podemos anotar los métodos que sean susceptibles de controlar las peticiones que se le realizan, tal y como se aprecia en el fragmento 9.7. Nótese que para el resto de métodos, la lógica es similar.

```
@GET
@Path("{id}")
@SLA()
@Produces(MediaType.APPLICATION_JSON)
public Bird find(@PathParam("id") Integer id) {
  return super.find(id);
```

Código 9.7: Extracto de un método de la API anotado con @SLA.

Por otro lado debemos inicializar el sistema cargando las plantillas y registrando los usuarios de prueba con sus correspondientes acuerdos, esto es un código muy similar al que mostramos en la sección anterior (fragmento 9.3 y C.17 del Anexo).

Finalmente, puesto que disponemos del código de ambos proyectos, modificamos el código del fichero de configuración (*Configuration.properties*) para adaptar los nombres a los usemos en el acuerdo.

Una vez hecho eso, tenemos una API guiada por SLAs en los métodos que hayamos anotado.

9.4 Comparativa

Hemos visto que podemos solucionar el problema de forma programática sin necesidad de usar acuerdos a nivel de servicio, aunque también se proponen dos soluciones usando una biblioteca externa que facilita la tarea. Llegados a este punto conviene analizar todos los casos y compararlos de forma objetiva.

Veamos un resumen de los pasos a dar en cada uno de los casos estudiados antes:

- **Implementación base**:
  1. Persistir de alguna forma el par token-peticiones.
  2. Crear filtro que compruebe si el número de peticiones del usuario ha sido excedido.
  3. Actualizar número de peticiones y persistir.
- **Implementación basada en AML**:
  1. Definir acuerdos y formalizarlos en iAgree o WS-Agreement.
  2. Instanciar AML por defecto o con configuración personalizada.
  3. Cargar los acuerdos y crear sistema de asignación de estos al usuario.
  4. Crear filtro que evalúe los *términos de garantía* deseados del acuerdo.
  5. Actualizar la *Monitorable Property* correspondiente a las peticiones.
- **Implementación basada en AGL**:
  1. Definir acuerdos y formalizarlos en iAgree o WS-Agreement.
  2. Importar proyecto AGL al proyecto base en JAX-RS.
  3. Anotar métodos que vayan a ser monitorizados por el filtro con @SLA.
  4. Adaptar el archivo de configuración a la nomenclatura del acuerdo.

  La opción más rápida quizás sea no emplear ningún tipo de acuerdo y codificar la lógica manualmente. Esto es adecuado si se trata de una API sencilla o el plan de precios es muy simple; sin embargo, en cuanto la complejidad o el tamaño aumenta, la mejor forma de mantener el sistema escalable quizás sea desacoplando la lógica de la autorización de peticiones de la de la API. Puesto que los SLAs son ya ampliamente usados, con un esfuerzo inicial al realizar la formalización, obtenemos grandes ventajas.

Parte VI

Manuales

10

Manual de uso de AGL

It's very simple. Look, scissors cuts paper. Paper covers rock. Rock crushes lizard. Lizard poisons Spock. Spock smashes scissors. Scissors decapitates lizard. Lizard eats paper. Paper disproves Spock. Spock vaporizes rock. And as it always has, rock crushes scissors.

Sheldon Lee Cooper,
Fictional character on The Big Bang Theory

Una vez detallado la arquitectura y los detalles de implementación de la solución propuesta, el propósito de este capítulo no es más que dar al lector las pautas necesarias para conseguir replicar el proyecto funcionalmente.

10.1 SITUACIÓN INICIAL

Para hacer uso de la solución que proponemos en este proyecto es preciso disponer de una API REST diseñada usando JAX-RS funcionando. Por tal motivo, asumamos que tenemos una sencilla API, denominada *PapamoscasAPI*[1], que muestra información sobre avistamientos de pájaros. En la figura §10.1 vemos la apariencia del portal inicial y en §10.2 el resultado de una petición GET sobre la API[2].

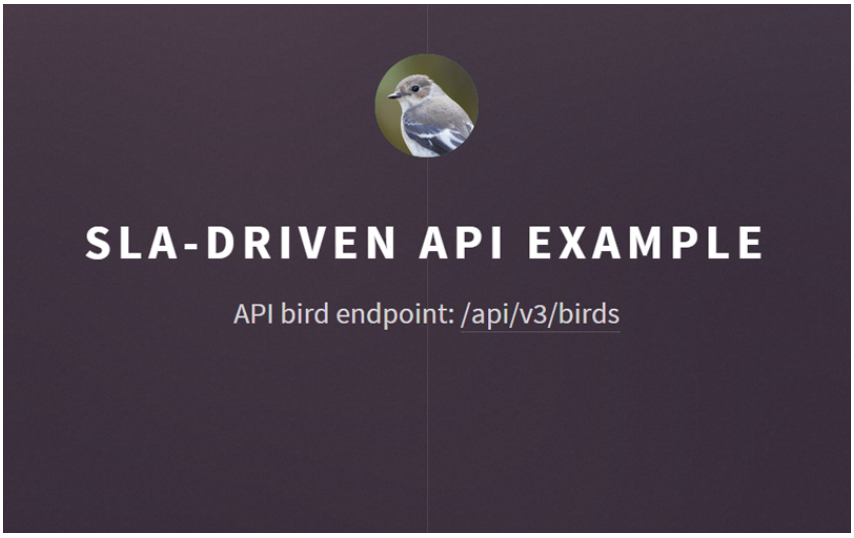

Figura 10.1: Vista de la pantalla inicial de la API

[1]A modo de curiosidad, un papamoscas (*Muscicapidae*) es un tipo de ave insectívora que se puede encontrar por ejemplo, en el Parque Natural de Doñana.

[2]El resultado del caso de ejemplo se encuentra desplegado en la nube.

```
[
  - {
        id: "1",
        specie: "papamosca",
        place: "roble",
        legDiameter: 0.1,
        wingSize: 10,
        eggs: 10,
        hatches: 3
    },
  - {
        id: "2",
        specie: "vencejo",
        place: "pino",
        legDiameter: 0.15,
        wingSize: 9,
        eggs: 7,
        hatches: 5
    },
  - {
        id: "3",
        specie: "papamosca",
        place: "pino",
        legDiameter: 0.1,
        wingSize: 10,
        eggs: 10,
        hatches: 3
    },
  - {
        id: "4",
        specie: "vencejo",
        place: "roble",
        legDiameter: 0.15,
        wingSize: 9,
        eggs: 7,
        hatches: 5
    }
]
```


Figura 10.2: Ejemplo de petición GET

A continuación debemos definir los planes de precio que vamos a aplicar en nuestra API. En concreto, *PapamoscasAPI* cuenta con tres planes: básico, medio y avanzado. Un usuario que se suscriba a ellos tendrá un máximo de 10 y 100 peticiones para los dos primeros casos e infinitas para el último. En la figura §10.3 se puede ver el plan de precios de esta API.

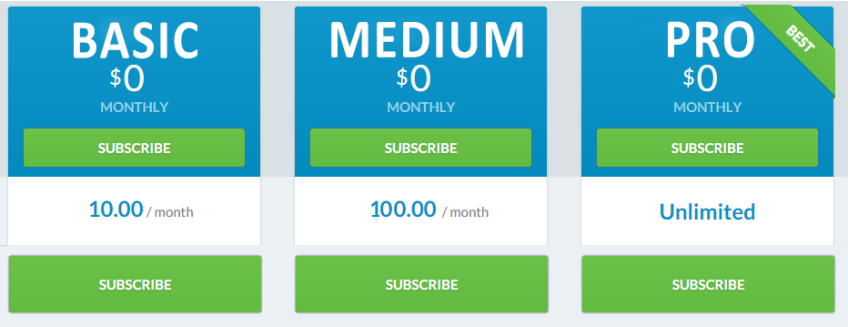

Figura 10.3: Plan de precios de *PapamoscasAPI*

10.2 DEFINIENDO EL ACUERDO

Ahora ofrecemos el esquema del fragmento 10.1 para generar una plantilla de acuerdo. Por cada plan que se ofrezca a los clientes de la API, se debe generar uno distinto.

```
1  Template IDENTIFICADOR version 1.0
2    Initiator: "NOMBRE DEL INICIADOR DEL SERVICIO";
3    Provider "NOMBRE DEL PROVEEDOR" as Responder;
4    Metrics:
5      int: integer[0..10000]; //rango a elegir
6  AgreementTerms
7    Service NOMBRE DEL SERVICIO availableAt "ENDPOINT DEL SERVICIO"
8      GlobalDescription
9    MonitorableProperties
10     global:
11       REQUEST_VAR: int = 0;
12   GuaranteeTerms
13     GT_NAME: Consumer guarantees REQUEST_VAR <=MAX_REQUESTS; //
     ↪ MAX_REQUESTS es fijo
14 EndTemplate
```

Código 10.1: Estructura de acuerdo.

En este sencillo ejemplo, tenemos los planes *basic plan*, *medium plan* y *pro plan*. Como sabemos por diferentes artículos, podemos lo convertir muy fácilmente a otros lenguajes más conocidos como *WS-Agreement*, cuya especificación está definida en [1], o incluso a [5]. El acuerdo que corresponde al *basic plan* formalizado en *WS-Agreement* se encuentra en el C.15 del Anexo.

La formalización en *iAgree* [13] del *basic plan* se puede observar en 10.2. Los planes *medium plan* y *pro plan* formalizados se encuentran, respectivamente, en C.13 y C.14 del Anexo.

Nótese que hemos modelado una plantilla de acuerdo y hemos añadido una propiedad *price*, sin embargo, el filtro de la solución no lo tendrá en cuenta. Además es importante destacar los nombres de:

- **MonitorableProperty**: *Requests*.
- **GuaranteeTerm**: *RequestsTerm*.

- **SLO**: *Requests* <= *10*.

```
1  Template basicPlanT version 1.0
2   Initiator: "Papamoscas SL";
3   Provider "Papamoscas SL" as Responder;
4   Metrics:
5   price: integer [0..500];
6   int: integer[0..10000];
7   AgreementTerms
8   Service BirdAPI availableAt
      ↪ "http://papamoscas-isa.appspot.com/api/v3/birds"
9    GlobalDescription
10   PlanPrice: price = 9;
11   MonitorableProperties
12   global:
13    Requests: int = 0;
14   GuaranteeTerms
15   RequestTerm: Consumer guarantees Requests <=10;
16 EndTemplate
```

Código 10.2: Plantilla del acuerdo para el *basic plan*

Hemos hablado de plantillas de acuerdo, pero éstas no llegan a ser un acuerdo completo todavía. El siguiente paso es que cada cliente presente la oferta de acuerdo con las variaciones que permitan los *CreationConstraints* de la plantilla. En este sencillo ejemplo, al no existir restricciones sobre la creación de ofertas, se puede afirmar que la plantilla ya es, en sí, un acuerdo. Lo único que habrá que hacer será registrar una copia de la plantilla por cada usuario dado de alta, siempre según el plan que hubiese contratado, cosa que haremos en otro paso de este manual.

10.3 IMPORTANDO Y CONFIGURANDO EL PROYECTO

Una vez que obtenemos el código del proyecto y lo importamos en el IDE, tendremos una jerarquía de directorios como la que se muestra en la Figura §10.4.

A continuación, es preciso añadir la dependencia Maven correspondiente, para que *PapamoscasAPI* importe el artefacto que genera el proyecto. Esto se realiza añadiendo las líneas que se observan en el Fragmento

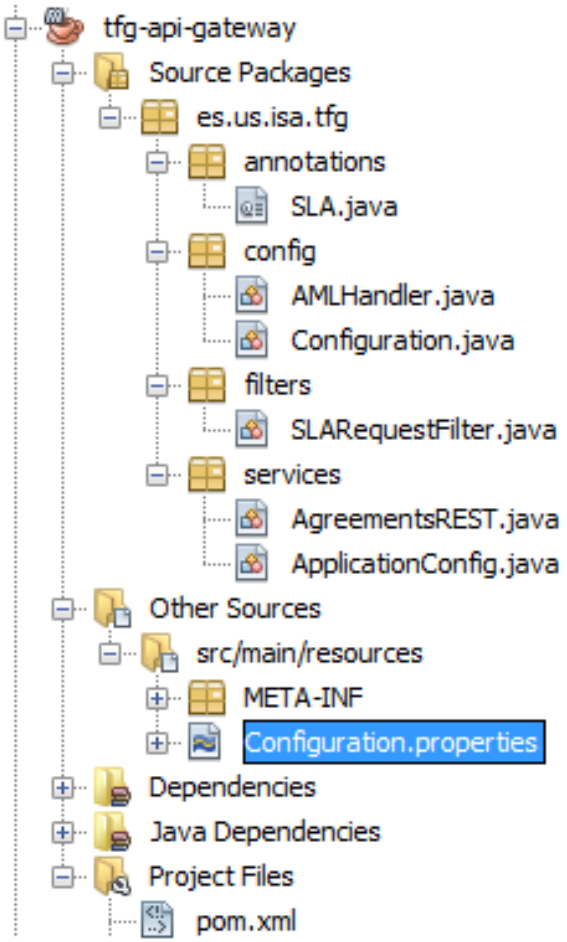

Figura 10.4: Jerarquía de directorios en el proyecto de la solución

```
1 <dependency>
2   <groupId>${project.groupId}</groupId>
3   <artifactId>tfg-api-gateway</artifactId>
4   <version>${project.version}</version>
5 </dependency>
```

Código 10.3: Extracto del POM del proyecto *PapamoscasAPI*.

La última tarea de configuración antes de poder usar la biblioteca en nuestra API es adecuar el contenido del archivo *Configuration.properties* según el acuerdo que hayamos definido. En particular, para la plantilla modelada en 10.2, tenemos el archivo mostrado en el Fragmento 10.4.

```
1 requestVar=Requests
2 requestGT=RequestTerm
```

```
3 queryParamToken=user
```

Código 10.4: Ejemplo de configuración para el acuerdo de *PapamoscasAPI*.

10.4 USANDO @SLA EN LA API

Con esto, ya podemos anotar los métodos que sean susceptibles de controlar las peticiones que se le realizan, tal y como se aprecia en el fragmento 10.5. Nótese que para el resto de métodos, la lógica es similar.

```
1 @GET
2 @Path("{id}")
3 @SLA()
4 @Produces(MediaType.APPLICATION_JSON)
5 public Bird find(@PathParam("id") Integer id) {
6   return super.find(id);
```

Código 10.5: Extracto de un método de *PapamoscasAPI* anotada con @SLA.

Por otro lado debemos debemos inicializar el sistema cargando las plantillas y registrando los usuarios de prueba con sus correspondientes acuerdos, esto es un código muy similar al que ya hemos visto. Se puede encontrar en el C.24 del Anexo.

10.5 PROBANDO EL SISTEMA

Una vez que hemos seguido todos pasos anteriores, podemos comenzar a hacer pruebas con diferentes usuarios y SLAs que definamos. En particular, en este ejemplo se han creado dos usuarios de cada tipo; además las cuotas por defecto de cada usuario son las mostradas en la Figura §10.5.

1. A continuación realizamos una petición de tipo GET sobre el recurso LOCALHOST:PORT/API /V3/BIRDS? USER=PROUSER1. Si antes no hemos hecho POST el resultado será vacío, pero en cualquier caso, se sirve la petición.

2. Ahora repetimos el caso de prueba, pero cambiando el token de usuario por *basicUser1*. Realizamos diez peticiones con este usuario y, a la undécima vez, un código de error nos es enviado, tal y como se muestra en la figura §10.6

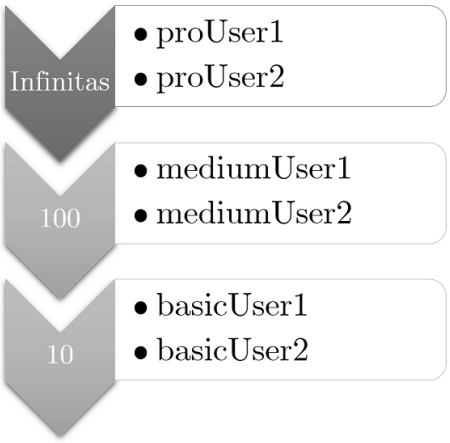

Figura 10.5: Usuarios y cuotas por defecto.

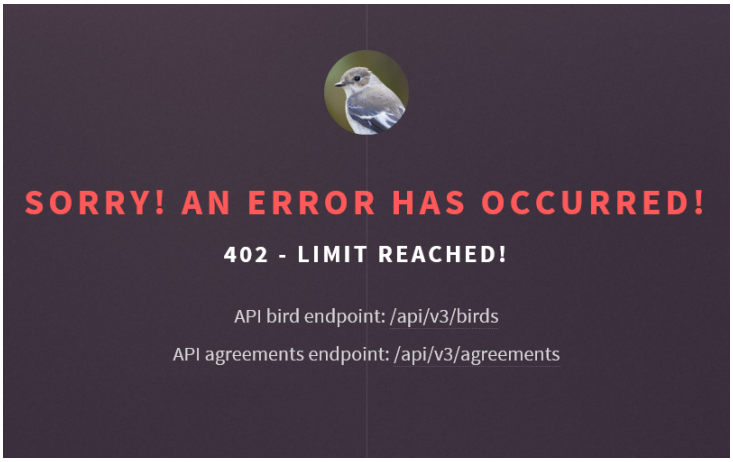

Figura 10.6: Error mostrado al alcanzar el límite.

Habiendo realizado lo anterior de forma satisfactoria, veamos qué partes podemos cambiar con facilidad para obtener un funcionamiento distinto.

- En primer lugar podríamos editar las plantillas de acuerdos. Actualmente hay tres planes que se pueden alterar como se desee antes de desplegar la aplicación. En concreto, podemos cambiar el *Guarantee Term RequestTerm: Consumer guarantees Requests <=10;* y así controlar el número de peticiones deseado.
- Otro aspecto fácil de modificar es la inicialización de usuarios. Desde el método *loadTestAgreements()* podemos cambiarlo.
- En caso de querer añadir una plantilla de acuerdo más, basta con añadirlo a la carpeta, cargarlo y luego añadir un usuario como arriba se comenta.

Parte VII

Conclusiones

Conclusiones

Space... the final frontier.

William Shatner (1966),
Star Trek

Llegados al final de este trabajo de fin de grado conviene realizar un análisis del proyecto, por ello, en el presente capítulo se realiza una evaluación final del mismo, destacando los hechos relevantes durante el transcurso del mismo. Finalmente se proponen y analizan diversas propuestas de extensiones o líneas de trabajo futuras.

11.1 Evaluación final del proyecto

Si queremos diseñar una API para comercializarla en mercado se han de resolver una serie de problemáticas como la gestión, control y limitación del número de peticiones realizadas a una API. Tras un análisis del estado del arte en el contexto de APIs y *Software as a Service* que nos llevó a un estudio de las soluciones de *API Gateway* que existen en el mercado actualmente, concluimos que no existía ninguna solución basada en acuerdos a nivel de servicio explícitos. El hecho de obtener una API guiada por SLAs nos permite desacoplar la lógica de negocio de lo fundamental de la aplicación, cosa que favorece la evolución del sistema y facilita el mantenimiento.

El presente trabajo de fin de grado se enmarca dentro de las líneas investigadoras del grupo de *Ingeniería del Software Aplicada* (ISA), en concreto, en un proyecto de gobernanza de SaaS y API en entornos *cloud* con soluciones basadas en SLAs en el que se ha seguido un proceso iterativo para conseguir los resultados planificados.

Desde el punto de vista de los *objetivos técnicos*, estos han sido completamente satisfechos, pues hemos realizado un estudio del arte en el contexto de APIs y *Gateways* y hemos desarrollado una prueba de concepto funcional y extensible. Por otro lado, los *objetivos académicos* que se habían plantando han sido igualmente satisfechos, pues ya existe dentro del grupo una visión a corto plazo de avanzar y mejorar los resultados que se han alcanzado en este proyecto.

Gracias a este proyecto se ha desarrollado un artículo para un congreso nacional de referencia en ingeniería de servicios (*Jornadas de Ciencia e Ingeniería de Servicios*, JCIS), cosa que ha supuesto la primera producción investigadora del autor del presente trabajo de fin de grado.

11.2 Trabajo futuro

En este trabajo se muestra una prueba de concepto que demuestra que el uso de un acuerdo formalizado puede ser usado para tareas de nivel operativo, por ejemplo, para autorizar peticiones y dar así soporte a los diversos modelos de cobro que puedan existir. Por ello, una primera vía de actuación sería seguir investigando en esta línea a fin de desarrollar una biblioteca de gestión de APIs mucho más versátil y con mayor funcionalidad.

Por otro lado, puesto que limitamos que la API esté basada en JAX-RS, una posible mejora tecnológica sería dar soporte a APIs realizadas con Spring en vez de JavaEE. De este modo se cubriría gran parte del mercado de APIs desarrolladas en Java.

Otra mejora tecnológica podría consistir en añadir un cierto *paylaod* a la anotación, de forma que cada método pudiera especificar modificadores que afectaran en el cómputo de las propiedades y términos de garantía. Por ejemplo, con *@SLA(cost*=2) podemos definir que el uso de ese recurso consume dos unidades en vez de una.

A nivel de investigación en el marco de los acuerdos a nivel de servicio ya existen líneas de trabajo activas y hay diversas actuaciones planificadas relacionadas con este proyecto.

Parte

Anexos

Towards SLA-Driven API Gateways

A.1 Abstract

[1]As APIs are becoming popular to build *Service-Based Applications* (SBA), API Gateways are being increasingly used to facilitate API features management. They offer API management functionality such as pricing plans support, user authentication, API versioning or response caching. Some parts of the information that an API Gateway needs are already included into a *Service Level Agreement* (SLA), that providers use to describe the rights and the obligations of involved parties in the service. Unfortunately, current API Gateways do not use any SLA representation model nor SLA underlying technology, thereby missing potential opportunities. In this paper we analyze the state of the art to justify the current situation and we identify some research challenges so as to achieve SLA-Driven API Gateways.

A.2 Introduction

As software architecture is shifting form a *product paradigm* to a *service paradigm* (i.e. *software as a service* (SaaS)), the traditional models, where users buy a perpetual-use license (e.g. *Microsoft Office*) are moving to service-oriented models, where users have to buy a subscription from the service provider (e.g. *Zoho*). Moreover, in the current *Service-Oriented Computing* paradigm, *Service-Based Applications* (SBA) are becoming increasingly important, especially nowadays with the popularization of cloud platforms.

Furthermore, we can find a large API marketplace where providers get paid for their clients API usage. In this context, most APIs have different pricing plans in order to fit with the customer's needs and they have to determinate the proper service level for each plan. These decisions need to be managed and aligned with business goals.

[1]Este artículo (con formato LNCS) será será presentado en las *XI JORNADAS DE CIENCIA E INGENIERÍA DE SERVICIOS*. En él se hace uso de la investigación desarrollada en las secciones §4 y §4.3.10.

Traditional service-oriented industries define contractual documents known as *Software Level Agreement* (SLA) which stipulate and commit the provider to a required level of service according their business model. We can extend SLA model to APIs providers so that we get SLA managed services where both customer and provider agree certain service properties, such as service level objectives, service features, up-time guarantees or number of requests.

Most API Gateways do not have any associated SLA, hence a rigid and complex business model evolution and a code difficult to maintain, thus created solutions will not be generic, scalable nor domain-independent ones. On the contrary, by having an implicit SLA, open, scalable, flexible and domain-independent solutions can be built.

When API providers try to align technology with their business goals they can use an API Gateway solution, i.e. a third-party solution that try to resolve some API management common tasks such as user authentication, establishing a certain request limit per user or blocking resources depending upon the user who called the API. This solution is uncoupled from the API code. Nevertheless, the service provider depends on a certain third-party gateway, which offers limited features that may not fit with the business needs.

In this work we analyze APIs and API Gateways to identify requirements for a future framework that supports SLA-driven API Gateways solutions. We analyze a few APIs and study what they have in common. Next, we focus on some API Gateway solutions existing in the market and we detail what they really offer. Finally, with this information, we could establish research goals so that we continue working on this line. Figure §A.1 shows a context diagram that represents the relationships between the elements we describe on the paper.

This paper is organized as follows: Section §A.3 describes the current situation of APIs by giving examples of reals ones and analyzing its features. In Section §A.4, we study a few examples of API gateways focus on its relationship with SLA management. Next, in Section §A.5 we try to identify new research challenges that can be used as requirements for a further design and development. Finally, Section §A.6 shows some remarks and conclusions.

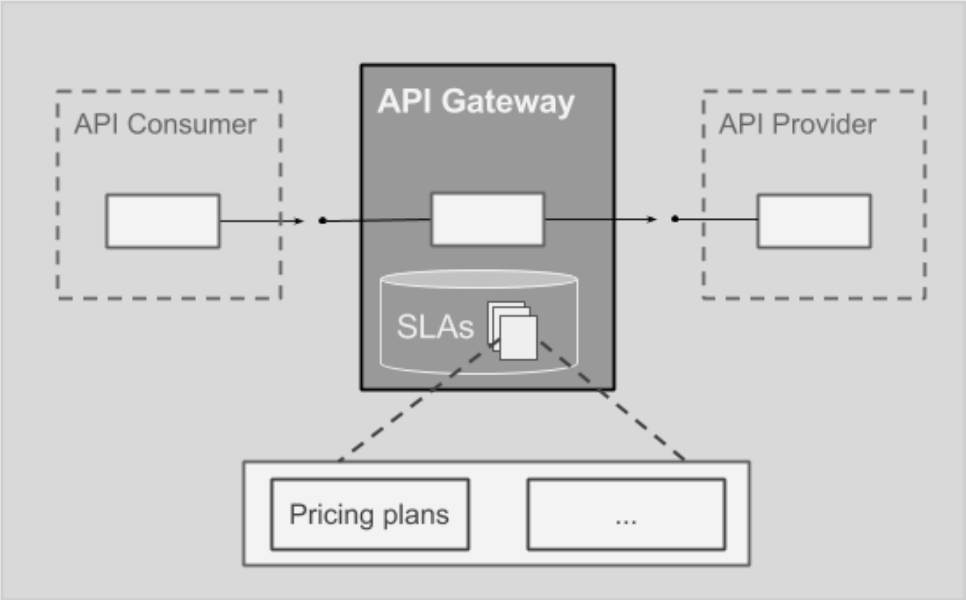

Figura A.1: Context diagram.

A.3 PRICING PLANS IN APIS

In this section we show the relationship between the properties that API providers are used to monitor and we analyze some real world APIs in order to highlight different pricing plans.

A Service Level Agreement (SLA) can be seen as containers of the functional and non-functional properties that both parties (the service consumer and the service provider) agree specifying its rights and obligations during the interaction. SLAs are widely used in the industry in situations where consumers and providers need or desire to explicitly express certain guarantees over the service transaction. There exists several purposes to describe SLAs, for instance, WS-Agreement [1], purposed by *Open Grid Forum*. It is one of the firsts specification which counts with well-defined validity criteria and an environment to edit, validate and monitor SLAs[2] [9].

In connection with SLAs, APIs often monitor properties for each user or each request (e.g. the number of requests, the called endpoint, user authentication data, etc.). According to these properties and their own business model, providers can design pricing plans and get paid for their service usage. An important challenge for API service providers is managing different plans with their customers while satisfying their business objectives. These monitorable properties could be introduced into the existent SLA between the consumer and the provider. For this reason, it is possible to establish

[2]This environment is deployed at isa.us.es/IDEAS.

a relationship among SLAs and pricing plans.

In order to illustrate the variability of existing API pricing plans we show and detail a real example in Figure §A.2.

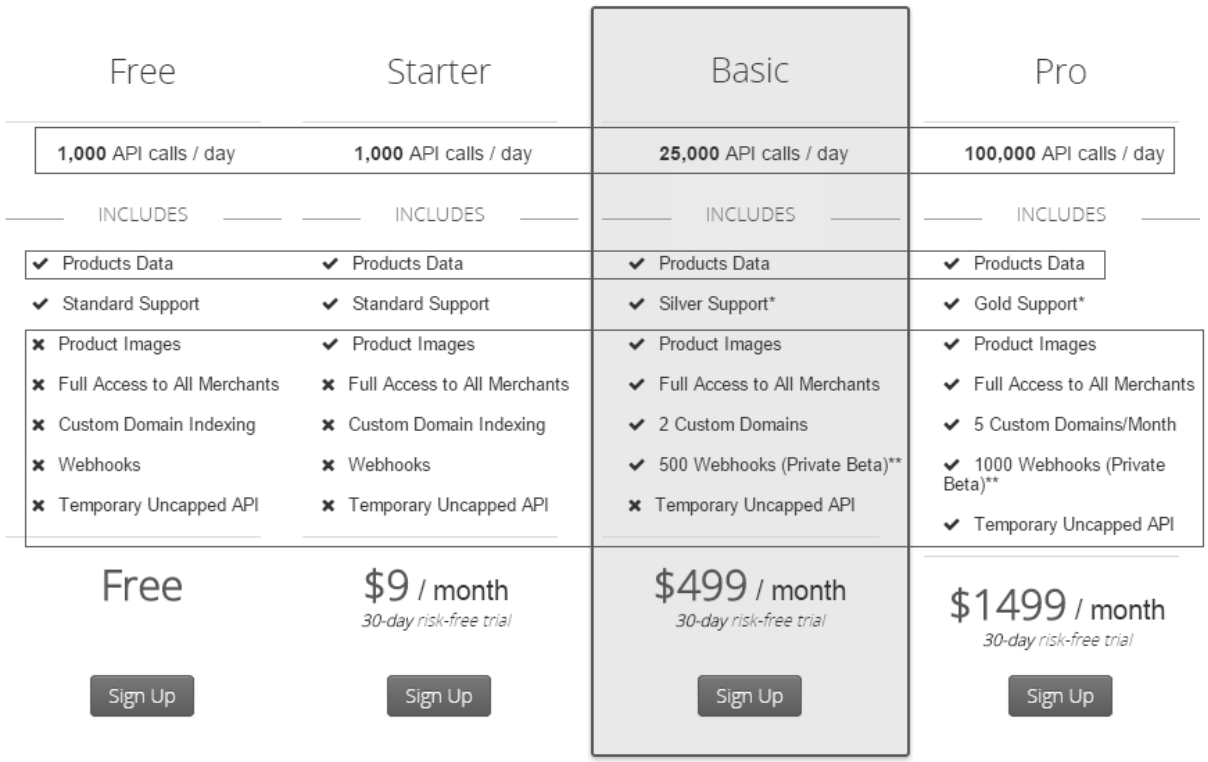

Figura A.2: Semantics3 Pricing Plan.

In Figure §A.2 we show the pricing plan of Semantic3 company, whose API provides access to a database of normalized e-commerce product, UPC, and pricing data. There exists 4 different plans which are primarily based upon daily requests amount. It also offers diverse features and support level for each plan. For instance, in *free plan* we are not able to access the full API data nor indexing custom domains. Definitely, most APIs have to control user requests and the features they offer to each client.

API pricing plans range from completely free to paid plans. In this context, in order to capture the variability in the different scenarios we have studied 9 different APIs.

- **Freemium (P_1)**, when there exists a limited basic plan and users can overtake the limits depending on they pay for the service, e.g. AlchemyAPI has free plan of 1 thousand transactions per day but subsequent ones are paid.

- **Tiered (P_2)**, when each tier has its own set or services and allowances for access to API resources and it is fixed a prize for each tier, e.g. Semantics3 has 3 different paid plans.
- **Unit-based (P_3)**, when pricing is based upon pre-defined unit of measurement, such as API call or storage on disk, e.g. GroupDocs establishes a base price and $0.15 is billed for each extra operation.
- **Pay-as-you-go (P_4)**, when a costumer accumulate a monthly bill basing on what is its use of API resources, e.g Stupeflix defines a cost per each API resource (such as $0.01 per minute of input video), thus the bill depends upon the user usage.
- **Enterprise pricing (P_5)**, when pricing is part of a larger sales process, it is a common way for large companies to price products and services based upon customer's needs, e.g. in AlchemyAPI, clients with over 1 billion requests per month are eligible for a custom plan.
- **Volume pricing (P_6)**, when pricing is based upon volume purchases by buying in bulk, e.g. MailGun defines an initial price of $0.00050 for first 500,000 sent emails and $0.00035 for the next 1,000,000.

We study some selected APIs in terms of their pricing plan they offer and additional features. In Figure §A.3 we illustrate that comparison[3], where a cell C_{ij} marked with an x means that API i supports the property j. Note there is not any property that could be considered as a particular case from another. Next, we detail an example extracted from the same table.

A large part of studied APIs have a unit-based (P_3) plan, probably due to it is easy to implement and is enough for many API providers. As an example, *AlchemyAPI* has two different pricing plans. On the one hand, it has 3 plans only based on the daily request volume (between 90.000 and 3 million) and a free tier of 1000 requests is also available. On the other hand, we could select a pay-as-you-use plan so that we would be billed only for the requests made.

It is important to separate how the total cost is calculated from how the client is billed. For instance, *Algorithmia* has a credit based plan, where it is necessary to add funds to one's account to be able to continue making API requests. Nevertheless, it is only a particular case of a unit-based plan.

[3]The reference of all studied APIs can be found at: goo.gl/0n8cPr

API Study						
API	P_1	P_2	P_3	P_4	P_5	P_6
AlchemyAPI	x	x	-	x	x	-
Algorithmia	-	-	-	x	x	-
FlightStats	-	-	-	-	x	x
Groupdocs	x	x	x	-	x	-
Kraken	-	x	x	-	-	-
Mailgun	x	-	-	x	x	x
OpenWeatherMap	x	x	-	-	-	-
Semantics3	x	x	-	-	x	-
Stupeflix	-	-	-	x	-	x

Figura A.3: APIs comparative.

As another example of unit-based plan, *FlightStats* has plan based upon requests you made, although it distinguishes according to resource type you call in the request (e.g. is more expensive to serve a request that implies more CPU or memory consumption). *Groupdocs*, whose pricing plan is not based upon the requests amount but predefined unit of measurement based their own business logic.

A.4 Pricing plans in API Gateways

In this section we discuss about the concept of *API gateways* (other authors already use this term: [18], [7]) and study how they could be related with pricing plans and SLA-Driven APIs by analyzing what they really offer.

An API gateway offers API management features, such as security (e.g. access control, DoS attacks blocking), pricing plans support, API analytics, request monitoring, API lifecycle control (e.g. service orchestration, govern data flows, API design support) or response transformation (e.g. JSON to SOAP). Furthermore, API Gateways be implemented as virtual appliances, virtual machine images, SaaS or reverse proxies.

From the 13 API Gateways studied we have selected 13 criteria in order to compare all of them. In Figure §A.4 we illustrate that comparison[4], where a cell C_{ij} marked with an x means that platform i has the property j, and we provide some examples extracted from the same table.

- **Security:**
  - **Basic Authentication (Q_1)**: it supports basic authorization methods (e.g. *Mashape let us configure authentication by a HTTP header, a query parameter and an API key*).
  - **Advanced Authentication (Q_2)**: it supports advanced authorization methods (e.g. *Apigee offers OAuth and LDAP authentications*).
  - **Attacks protection (Q_3)**: it includes a way to protect itself from attacks (e.g. *CA prevents our API from DoS attacks*).
- **Pricing plans support:**
  - **Free Tier (Q_4)**: it has a free plan with basic features or limited by somehow (e.g. *3Scale offers a basic tier with 50 thousand calls per day for free*).
  - **Requests limit (Q_5)**: there exists a way to establish quotas per user (e.g. *with Azure it is possible to limit API calls by user to a custom value*).
  - **Enterprise Plans (Q_6)**: it offers customized plans for enterprise clients (e.g. *3Scale let enterprise clients configure unlimited APIs with a guaranteed availability*).
- **Lifecycle control:**
  - **API versions (Q_7)**: it supports versioning (e.g. *Axway let us set different versions from the same API*).
  - **Orchestration (Q_8)**: there exists a way to orchestrate RESTful services in terms of SOA governance (e.g. *Akana let us control complete API lifecycle and orchestrate all services involved*).
- **Other features:**

[4]The reference of all studied API Gateways can also be found at: goo.gl/0n8cPr

- **OpenSource (Q_9)**: the code is available to anyone (e.g. *WSO2 and API Umbrella are both Open Source solutions*).
- **Load balancing (Q_{10})**: it makes possible to do load balancing between various API servers (e.g. *API Umbrella balance the requests between our backend*).
- **Cache (Q_{11})**: it includes request caching in order to reduce the effective API calls amount (e.g. *Apigee save the API responses in order to serve the same data if there is no changes*).
- **Response conversion (Q_{12})**: it gives the possibility of converting API response into another language (e.g. *Akana perform a JSON to SOAP conversion*).
- **Documentation (Q_{13})**: it generates a developer portal with API documentation (e.g. *Mashape generate an interactive documentation and a playground where developers can make requests*).

3Scale is an API gateway service provider who let us connect up to 3 different APIs with a daily traffic close to one million requests. It is possible to establish requests filtering policies based upon trusted API keys, black and whitelisting or OAuth authentication. In order to create billing plans, we could limit the requests amount per user and create a billing for that use or simply a billing plan based upon the requests that a user has made; it is also possible to create trial plans. Besides these core features, it also offers several statistics, a developer portal, documentation generation and different ways to deploy the reverse proxy towards *3Scale*.

Almost all API gateway studied services offer the same core functionality: request authorization and quota establishment so that various billing plans could be created. It might be enough for few APIs that do not need a more complex business billing logic, but does not allow to introduce all business logic we may need.

A.5 Research Challenges

We propose API Gateways based on SLAs. We have analyzed API properties and billing needs and how third-party API Gateways solutions could be useful to solve this problem. Nevertheless, there are some API requirements that are not satisfied by the previous solution, so the only possibility is to code it manually into the API code. This implies an increasing error probability and a coupling API code itself with API

API *Gateways* Study													
Gateway	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}
3Scale	x	-	-	x	x	x	-	-	-	-	-	-	x
Akana API Gateway	x	x	x	-	x	-	-	x	-	-	-	x	-
API Umbrella	x	-	-	x	-	-	-	-	x	x	x	-	x
Apiaxle	x	-	-	x	-	-	-	-	x	-	-	-	-
Apigee Edge API	x	x	x	-	x	x	x	-	-	-	x	-	x
Axway API Gateway	x	x	-	-	x	x	x	x	-	x	x	x	-
Azure API Management	-	-	-	-	x	-	-	-	-	-	-	x	x
CA API Gateway	x	x	x	-	-	-	x	-	-	-	x	-	-
Mashape	x	-	-	x	x	x	-	-	-	-	-	-	x
Mashery API Gateway	x	-	-	-	x	-	-	-	-	-	-	-	x
Monarch API Manager	x	-	-	x	-	-	-	-	x	-	-	-	-
Repose	x	-	-	x	x	-	-	-	x	x	-	-	-
WSO2 API Management	x	x	-	x	x	x	x	-	x	-	-	x	x

Figura A.4: API Gateways comparative.

management one. Previously we have shown the importance of considering SLAs in our design, therefore we focus on a SLA-Driven solution.

In this section we analyze these requirements and we try to identify other ones that could appear in a different scenario. Finally, we propose several research challenges in order to establish a road-map to focus our researches on.

We can identity new requirements that could be met by a theoretical API that use a SLA based pricing plan.

- **Challenge 1**: supporting temporarily in pricing plans. As current API gateways are static, if an API pricing plan depends on the moment that the API is called, it would be necessary to code this logic manually. It is possible to find more examples, such as support plan life-time (i.e. when it starts and expires) or time periods for guarantee terms (e.g. availability level is guaranteed only during Monday-

Friday working hours). In this way, we have dynamics and elastic pricing plans, regulated according to demand cycles. In [10], [12] and [11] authors analyze formal temporarily in SLAs.

- **Challenge 2**: adding a more fine-grained configuration by supporting other features and metrics than the requests amount. An API could define custom metrics that represent the cost for the business. For instance, metrics such as CPU time consumption, memory or storage usage.
- **Challenge 3**: adding support to compensations (penalties and rewards) according to the under-fulfillment or over-fulfillment of SLA service level objectives (SLO). For instance, API provider could be committed to serve requests in an established time; if there is a percentage of these request served out of time, provider should compensate client by certain way (e.g. claiming a refund). In [15] and [14] authors show the formal specification of a SLA with compensations.

A.6 Conclusions

In this paper we have shown an analysis of the state of art in API and API Gateway paradigm and propose a SLA-Driven solution in an API Gateway design. The API and SaaS management is a difficult task and there is no solution yet that help providers to automatize simple things like pricing management, feature limitation, billing based upon custom parameters, etc. Currently, service providers have to code this logic by hand or they ought to resort to third-party platforms, which involves an extra cost. We try to define the main research lines so that we could continue working on and, in the future, we could give a framework that permits APIs be managed by Service Level Agreements.

Memoria para la beca de iniciación a la investigación de la US

B.1 Introducción

El término *microservicio* ha ido adquiriendo notoriedad en un contexto de la Ingeniería del Software donde el paradigma de la Computación Orientada a Servicios cuenta mayor peso actualmente. Se observa que desde una década atrás se ha hablado de *arquitectura orientada a servicios (SOA)* y cómo sus principios aportan valor a la organización; dichos pilares de *SOA* incluyen el bajo acoplamiento, abstracción, reusabilidad, autonomía e interoperatibilidad. En definitiva, elementos imprescindibles para alejarse de las arquitecturas monolíticas convencionales.

Bajo la denominación de microservicio se incluyen desde servicios web de tipo *REST* o *SOAP* hasta nuevos modelos de distribución de software bajo demanda, conocidos como *SaaS*. De ahí que la proliferación de estos nuevos modelos han llegado a convertirlos en estándar *de facto* en el paradigma de la producción de software. En este contexto, las posibilidades de personalización y adaptación del servicio a cada cliente van más allá de la necesidad de integración y orquestación del servicio.

Históricamente en entornos empresariales se han extendido numerosas herramientas que ofrecen soporte a los principales pilares de una arquitectura orientada a servicios: integración, orquestación, enrutado, monitorización... casi todas ellas están bajo el término *Enterprise Service Bus (ESB)* o *Enterprise Application Integration (EAI)*. Actualmente las necesidades de la organización se mueven desde una integración centralizada hacia un contexto distribuido y flexible. Es por ello que ha existido un cambio de visión de la arquitectura orientada a servicios; los microservicios deben seguir dando soporte a ciertas necesidades, como la integración, publicación y descubrimiento de servicios, administración y despliegue escalable.

Casi todas estas necesidades están siendo ya cubiertas en mayor o menor medida, sin embargo, la gestión y automatización de contratos de servicios está aún sin resolver de una forma consensuada; entendiendo contrato como el compromiso y acuerdo alcanzados entre el consumidor y el productor del servicio. Para esta tarea cada organización emplea sus propias estrategias, pero casi todas confluyen en sistemas de monitorización ligados a un acuerdo en lenguaje natural.

En este contexto, el presente proyecto de investigación se enfoca en estudiar las formas de mejorar la gestión y automatización de *acuerdos a nivel de servicio (SLA)* en microservicios que permiten la personalización y adaptación al cliente y, en concreto, su relación con los nuevos modelos *Software as a Service (SaaS)*.

B.2 Background del equipo investigador

Este proyecto se enmarca dentro de las líneas investigadoras del grupo de *Ingeniería del Software Aplicada* (TIC205) de la Universidad de Sevilla, coordinado por el Dr. Antonio Ruiz Cortés; en un contexto de Líneas de Productos, Métodos Formales Aplicados a Familias de Procesos y de Productos, Ingeniería de Requisitos, Sistemas Complejos y, sobre todo, Computación Orientada a Servicios.

En concreto, el proyecto de investigación guarda estrecha relación con el proyecto financiado por el Ministerio de Economía y Competitividad *Tecnologías Avanzadas para Procesos como Servicios* (TIN2012-32273). Este proyecto incluye entre sus principales metas el soporte a servicios que puedan personalizarse de acuerdo a las necesidades del cliente y correr en la nube, un claro caso de necesidad de aplicación de acuerdos a nivel de servicio.

El grupo de investigación ha sido pionero y referente internacional en la formalización de acuerdos desde hace una década. En este marco, destacan artículos como [17], automatizando el análisis de conflictos de *SLA* tratándolo como *problemas de satisfacción de restricciones (CSP)*; [14], ofreciendo una explicación de las violaciones al acuerdo en tiempo de ejecución mediante una herramienta de monitorización; [15], dando un modelo para la creación de *SLA* con penalizaciones o recompensas según el cumplimiento de *objetivos de nivel de servicio (SLO)* y [4], dando herramientas para la edición y validación de documentos de acuerdos formalizados en lenguajes estándares como *WS-Agreement*.

Varias tesis doctorales han sido realizadas también en este campo, por ejemplo, [8], donde se estudia la relación entre servicios web y la programación con restricciones; y [13], donde se define un lenguaje alternativo para la formalización de acuerdos y se realiza la correspondencia con operaciones de análisis de tipo *CSP*.

B.3 Objetivos

Todos los objetivos del plan de trabajo van destinados a converger con la línea investigadora del grupo, además de pretender sentar una base sólida sobre la que poder continuar en futuras investigaciones. En concreto, definimos dos objetivos fundamentales:

1. Analizar el *estado del arte* en acuerdos de servicios reales.

2. Desarrollar los principales requisitos de un futuro *framework* para el gobierno de *SaaS*.

En primer lugar, se pretende realizar un estudio detallado de los acuerdos en servicios reales y mayoritariamente usados para así tener una visión más completa y detallada de las necesidades a satisfacer en fases más avanzadas del proyecto. Los servicios a analizar se dividen, principalmente, en dos tipos, las *APIs* y los proveedores de *SaaS*.

Por otro lado, y en relación con el segundo objetivo fundamental, se estudiarán las herramientas existentes en el mercado para proporcionar soporte a la gestión de microservicios. En particular, se verán distintos *proxies* de *APIs* que ofrecen valor al desarrollador y consumidor de la misma.

Toda esta investigación pretende confluir en la identificación de los requisitos que un *framework* para el gobierno de aplicaciones con modelo de *Software como Servicio* debe satisfacer.

Con esto se pretenden sentar unas bases sólidas sobre las que apoyar una futura investigación y desarrollo de la plataforma que ofrezca un soporte eficaz y logre solucionar los problemas y deficiencias actuales en el gobierno de las arquitecturas basadas en microservicios.

B.4 Plan de trabajo

Durante las 8 semanas durante los meses de julio y septiembre de 2015 se estima la siguiente planificación:

Resumen de la planificación	
Julio	
Semana 1	Introducción al contexto del problema.
Semana 2	Identificación de plataformas *SaaS* y *APIs* a estudiar.
Semana 3	Estudio de proveedores de *APIs*.
Semana 4	Estudio de proveedores *SaaS*.
Septiembre	
Semana 5	Recapitulación de resultados obtenidos.
Semana 6	Identificación de requisitos y necesidades primarias.
Semana 7	Análisis e identificación de requisitos secundarios.
Semana 8	Análisis final y valoración de resultados.

Cuadro B.1: Tabla resumen de planificación semanal.

La carga de trabajo se ha repartido a razón de 70/30 entre los dos objetivos fundamentales identificados, siendo la planificación coherente con tal reparto. No obstante, es susceptible de cambios en función de los resultados que se vayan obteniendo.

Códigos

```
1 < 13 > { } =[100%]=> < 13 > Analíticas;
2 < 13 > Analíticas =[92%]=> < 12 > BasiAuth;
3 < 9 > Cuotas Analíticas =[89%]=> < 8 > BasiAuth;
4 < 7 > FreeTier Analíticas =[100%]=> < 7 > BasiAuth;
5 < 7 > Analíticas Portal =[86%]=> < 6 > Cuotas;
6 < 7 > Analíticas Portal =[86%]=> < 6 > BasiAuth;
7 < 5 > OpenSource Analíticas =[100%]=> < 5 > FreeTier BasiAuth;
8 < 5 > AdvAuth Analíticas =[100%]=> < 5 > BasiAuth;
9 < 5 > Planes Analíticas =[100%]=> < 5 > BasiAuth Cuotas;
10 < 6 > Cuotas Analíticas Portal =[83%]=> < 5 > BasiAuth;
11 < 4 > Analíticas Versionado =[100%]=> < 4 > BasiAuth AdvAuth;
12 < 4 > Analíticas Conversión =[100%]=> < 4 > Cuotas;
13 < 4 > Analíticas Caché =[100%]=> < 4 > BasiAuth;
14 < 5 > BasiAuth Cuotas Analíticas Portal =[80%]=> < 4 > Planes;
15 < 5 > BasiAuth Planes Cuotas Analíticas =[80%]=> < 4 > Portal;
16 < 5 > BasiAuth AdvAuth Analíticas =[80%]=> < 4 > Versionado;
17 < 5 > BasiAuth AdvAuth Analíticas =[80%]=> < 4 > Cuotas;
18 < 3 > FreeTier BasiAuth Planes Cuotas Analíticas =[100%]=> < 3 >
   ↪ Portal;
19 < 3 > FreeTier BasiAuth Cuotas Analíticas Portal =[100%]=> < 3 >
   ↪ Planes;
20 < 3 > BasiAuth AdvAuth Planes Cuotas Analíticas =[100%]=> < 3 >
   ↪ Versionado;
21 < 3 > BasiAuth AdvAuth Cuotas Analíticas Versionado =[100%]=> < 3
   ↪ > Planes;
22 < 3 > BasiAuth AdvAuth Analíticas Caché =[100%]=> < 3 >
   ↪ Versionado;
23 < 3 > BasiAuth Cuotas Analíticas Conversión =[100%]=> < 3 >
   ↪ AdvAuth;
24 < 3 > Analíticas Protección =[100%]=> < 3 > BasiAuth AdvAuth;
25 < 3 > Analíticas Balanceo =[100%]=> < 3 > BasiAuth;
```

```
26 < 2 > FreeTier BasiAuth Analíticas Balanceo =[100%]=> < 2 >
   ↪ OpenSource;
27 < 2 > BasiAuth AdvAuth Analíticas Portal =[100%]=> < 2 > Planes
   ↪ Cuotas Versionado;
28 < 2 > BasiAuth AdvAuth Analíticas Versionado Protección =[100%]=>
   ↪ < 2 > Caché;
29 < 2 > BasiAuth Cuotas Analíticas Caché =[100%]=> < 2 > AdvAuth
   ↪ Planes Versionado;
30 < 2 > Analíticas Orquestación =[100%]=> < 2 > BasiAuth AdvAuth
   ↪ Cuotas Conversión;
31 < 1 > OpenSource FreeTier BasiAuth Planes Cuotas Analíticas
   ↪ Portal =[100%]=> < 1 > AdvAuth Versionado Conversión;
32 < 1 > FreeTier BasiAuth AdvAuth Analíticas =[100%]=> < 1 >
   ↪ OpenSource Planes Cuotas Portal Versionado Conversión;
33 < 1 > FreeTier BasiAuth Analíticas Caché =[100%]=> < 1 >
   ↪ OpenSource Portal Balanceo;
34 < 1 > BasiAuth AdvAuth Planes Cuotas Analíticas Portal Versionado
   ↪ Conversión =[100%]=> < 1 > OpenSource FreeTier;
35 < 1 > BasiAuth AdvAuth Planes Cuotas Analíticas Portal Versionado
   ↪ Caché =[100%]=> < 1 > Protección;
36 < 1 > BasiAuth AdvAuth Planes Cuotas Analíticas Versionado
   ↪ Conversión Caché =[100%]=> < 1 > Balanceo Orquestación;
37 < 1 > BasiAuth AdvAuth Planes Cuotas Analíticas Versionado
   ↪ Conversión Orquestación =[100%]=> < 1 > Balanceo Caché;
38 < 1 > BasiAuth AdvAuth Planes Cuotas Analíticas Versionado
   ↪ Protección Caché =[100%]=> < 1 > Portal;
39 < 1 > BasiAuth AdvAuth Cuotas Analíticas Conversión Protección
   ↪ =[100%]=> < 1 > Orquestación;
40 < 1 > BasiAuth AdvAuth Analíticas Balanceo =[100%]=> < 1 > Planes
   ↪ Cuotas Versionado Conversión Caché Orquestación;
41 < 1 > BasiAuth Planes Cuotas Analíticas Balanceo =[100%]=> < 1 >
   ↪ AdvAuth Versionado Conversión Caché Orquestación;
42 < 1 > BasiAuth Analíticas Portal Balanceo =[100%]=> < 1 >
   ↪ OpenSource FreeTier Caché;
```

Código C.1: Salida del análisis formal de conceptos sobre estudio de API *gateways*.

```
1 @ApplicationScoped
2 public class SingletonMapBean implements Serializable{
3   private Map<Object, Map<String, Object>> map = new HashMap<>();
```

```
4   public Map<String, Object> getMetadata(Object obj) {
5     return map.get(obj);
6   }
7   public Object getMetadataValue(Object obj, String property) {
8     return map.get(obj).get(property);
9   }
10  public void setMetadata(Object obj, String property, Object value)
      ↪ {
11    if (!map.containsKey(obj)) {
12      map.put(obj, new HashMap<String, Object>());
13      map.get(obj).put(property, value);
14    } else {
15      map.get(obj).put(property, value);
16    }
17  }
18 }
```

Código C.2: Clase principal de la biblioteca de prueba en JavaEE.

```
1  <dependencies>
2    <dependency>
3      <groupId>com.tfg</groupId>
4      <artifactId>BestMapLib</artifactId>
5      <version>1.0-SNAPSHOT</version>
6      <type>jar</type>
7    </dependency>
8    <dependency>
9      <groupId>javax</groupId>
10     <artifactId>javaee-api</artifactId>
11     <version>7.0</version>
12     <scope>provided</scope>
13   </dependency>
14   <dependency>
15     <groupId>org.jboss.weld.servlet</groupId>
16     <artifactId>weld-servlet</artifactId>
17     <version>2.2.6.Final</version>
18   </dependency>
19   <dependency>
20     <groupId>javax.servlet</groupId>
21     <artifactId>javax.servlet-api</artifactId>
```

```
22      <version>3.1.0</version>
23      <scope>provided</scope>
24    </dependency>
25  </dependencies>
```

Código C.3: Extracto del POM de la biblioteca de prueba en JavaEE.

```
1  @WebServlet(name = TestServlet, urlPatterns = {})
2  public class TestServlet extends HttpServlet {
3    @Inject
4    private HelloBean hb;
5    @Override
6    protected void service(HttpServletRequest req, HttpServletResponse
       ↪ res) throws IOException, ServletException {
7      PrintWriter writer = res.getWriter();
8      writer.println(JavaEE en proyecto JavaEE);
9      writer.println(HelloBean says  + hb.sayHello());
10   }
11 }
```

Código C.4: Servlet en el proyecto JavaEE.

```
1  import javax.enterprise.inject.Produces;
2  import com.tfg.SingletonMapBean;
3  //importado del JAR en nuestro repositorio local Maven.
4  public class SingletonMapBeanProducer {
5   @Produces
6   public SingletonMapBean singletonMapBeanProducer(){
7    return new SingletonMapBean();
8   }
9  }
```

Código C.5: Clase productora para poder inyectar la biblioteca en el proyecto JavaEE.

```
1  public class HelloBean implements Serializable {
2          @Inject
3          private SingletonMapBean metadata;
4    public String sayHello() {
5      Integer numero = new Double(10 * Math.random() + 1).intValue();
6      metadata.setMetadata(numero, "creado", new Date());
7      metadata.setMetadata(numero, "propietario", "antgamdia");
```

```
8       return metadata.getMetadata(numero);
9     }
10  }
```

Código C.6: Bean de JavaEE que usa la biblioteca de prueba en JavaEE.

```
1  <groupId>com.tfg</groupId>
2  <artifactId>mapLib_Spring</artifactId>
3  <version>1.0-SNAPSHOT</version>
4  <packaging>jar</packaging>
5  <dependencyManagement>
6    <dependencies>
7      <dependency>
8        <groupId>io.spring.platform</groupId>
9        <artifactId>platform-bom</artifactId>
10       <version>1.0.3.RELEASE</version>
11       <type>pom</type>
12       <scope>import</scope>
13     </dependency>
14   </dependencies>
15 </dependencyManagement>
16 <dependencies>
17   <dependency>
18     <groupId>org.springframework</groupId>
19     <artifactId>spring-core</artifactId>
20   </dependency>
21 </dependencies>
```

Código C.7: Extracto del POM de la biblioteca de prueba en Spring.

```
1  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
2  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
3  <beans>
4    <bean id="id1" class="SingletonMapBean" scope="singleton"/>
5  </beans>
```

Código C.8: Configuración XML de la biblioteca en Spring.

```
1  <dependencies>
2    <!-- Spring framework -->
3    <dependency>
```

```
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
    <version>2.5.6</version>
  </dependency>
  <!-- Spring MVC framework -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>2.5.6</version>
  </dependency>
  <!-- JSTL -->
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.1.2</version>
  </dependency>
  <!-- Apache Taglibs -->
  <dependency>
    <groupId>taglibs</groupId>
    <artifactId>standard</artifactId>
    <version>1.1.2</version>
  </dependency>
  <!-- Nuestra biblioteca -->
   <dependency>
    <groupId>com.tfg</groupId>
    <artifactId>mapLib_Spring</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>jar</type>
  </dependency>
</dependencies>
```

Código C.9: Extracto del POM del proyecto de prueba en Spring.

```
public class Controller extends AbstractController {
  private final SingletonMapBean metadata = new SingletonMapBean();
  @Override
  protected ModelAndView handleRequestInternal(HttpServletRequest
   ↪ request, HttpServletResponse response) throws Exception {
    ModelAndView model = new ModelAndView("index");
    model.addObject("msg", sayHello());
```

```
7       return model;
8     }
9     private String sayHello() {
10      Integer numero = new Double(10 * Math.random() + 1).intValue();
11      metadata.setMetadata(numero, "creado", new Date());
12      metadata.setMetadata(numero, "propietario", "antgamdia");
13      return metadata.getMap().toString();
14    }
15  }
```

Código C.10: Controlador del proyecto de prueba en Spring.

```
1   public class AgreementFilter_noAML implements Filter {
2     private static final Logger LOG = Logger.getLogger
        ↪ (AgreementFilter_noAML.class.getName());
3     private String clientId;
4
5     @Override
6     public void init(FilterConfig fConfig) throws ServletException {
7       ServletContext ctx = fConfig.getServletContext();
8     }
9
10    @Override
11    public void doFilter(ServletRequest request, ServletResponse
        ↪ response, FilterChain chain) throws IOException,
        ↪ ServletException {
12      HttpServletRequest req = (HttpServletRequest) request;
13      HttpServletResponse resp = (HttpServletResponse) response;
14      clientId = req.getParameter("user");
15      if (clientId != null && authorizeRequest(req)) {
16        requestDone(req);
17        chain.doFilter(request, response);
18      } else {
19        resp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
20        resp.sendRedirect("/error.jsp");
21      }
22    }
23
24    @Override
25    public void destroy() {
```

```
26
27    }
28
29    private boolean authorizeRequest(HttpServletRequest req) {
30      Integer availableRequests = Helper_noAML.getInstance().
       ↪ getAvailableRequestsFromUser(clientId);
31      return availableRequests > 0 || availableRequests == -1;
32    }
33
34    private void requestDone(HttpServletRequest req) {
35      Integer availableRequests = Helper_noAML.getInstance().
       ↪ getAvailableRequestsFromUser(clientId);
36      if (availableRequests > 0) {
37        availableRequests--;
38        Helper_noAML.getInstance().setRequests(clientId,
       ↪ availableRequests);
39      }
40    }
```

Código C.11: Clase AgreementFilter sin AML.

```
1  public class Helper_noAML {
2
3    private static Helper_noAML instance;
4    private Map<String, Integer> availableReqMap = new HashMap<>();
5
6    protected Helper_noAML() {
7    }
8
9    public static Helper_noAML getInstance() {
10     if (Helper_noAML.instance == null) {
11
12       Helper_noAML.instance = new Helper_noAML();
13
14       Helper_noAML.instance.availableReqMap.put ("basicUser1", 10);
15       Helper_noAML.instance.availableReqMap.put ("basicUser2", 10);
16
17       Helper_noAML.instance.availableReqMap.put ("mediumUser1", 100);
18       Helper_noAML.instance.availableReqMap.put ("mediumUser2", 100);
19
```

```
20        Helper_noAML.instance.availableReqMap.put ("proUser1", -1);
21        Helper_noAML.instance.availableReqMap.put ("proUser2", -1);
22      }
23      return Helper_noAML.instance;
24    }
25
26    public void setRequests(String clientId, Integer requests) {
27      Helper_noAML.instance.availableReqMap.put(clientId, requests);
28    }
29
30    public Integer getAvailableRequestsFromUser(String clientId) {
31      try {
32        return Helper_noAML.instance.availableReqMap.get(clientId);
33      } catch (Exception e) {
34        return 0;
35      }
36    }
```

Código C.12: Clase Helper sin AML.

```
1  Template mediumPlanT version 1.0
2   Initiator: "Papamoscas SL";
3   Provider "Papamoscas SL" as Responder;
4   Metrics:
5   price: integer [0..500];
6   int: integer[0..10000];
7   AgreementTerms
8   Service BirdAPI availableAt
      ↪ "http://papamoscas-isa.appspot.com/api/v3/birds"
9    GlobalDescription
10   PlanPrice: price = 99;
11   MonitorableProperties
12   global:
13    Requests: int = 0;
14   GuaranteeTerms
15   RequestTerm: Consumer guarantees Requests <=100;
16 EndTemplate
```

Código C.13: Plantilla del acuerdo para el *medium plan*.

```
1  Template proPlanT version 1.0
```

```
2  Initiator: "Papamoscas SL";
3  Provider "Papamoscas SL" as Responder;
4  Metrics:
5  price: integer [0..500];
6  int: integer[0..10000];
7  AgreementTerms
8  Service BirdAPI availableAt
     ↪ "http://papamoscas-isa.appspot.com/api/v3/birds"
9   GlobalDescription
10  PlanPrice: price = 999;
11  MonitorableProperties
12  global:
13   Requests: int = 0;
14  GuaranteeTerms
15  RequestTerm: Consumer guarantees Requests >= 0;
16 EndTemplate
```

Código C.14: Plantilla del acuerdo para el *pro plan*.

```
1  <?xml version="1.0" encoding = "UTF-8"?>
2  <wsag:Template wsag:TemplateId="1.0"
3  xmlns:xsi="http:// www.w3.org/2001/ XMLSchema-instance"
4  xmlns:wsag="http:// schemas.ggf.org/graap/2007/03/ ws-agreement"
5  xmlns:xs="http:// www.w3.org/2001/XMLSchema"
6  xmlns:iag="http:// www.isa.us.es/schemas/iagree"
7  xsi:schemaLocation="http:// .ggf.org/graap/2007/03/ wsagreement"
8  agreement_types.xsd="http:// www.w3.org/2001/ XMLSchema
     ↪ XMLSchema.xsd">
9    <wsag:Name>basicPlanT</wsag:Name>
10   <wsag:Context>
11     <wsag:AgreementResponder id="Papamoscas
     ↪ SL">Provider</wsag:AgreementResponder>
12     <iag:Metrics>
13       <iag:Metric id="price" type="integer"
     ↪ domain="[0..500]"></iag:Metric>
14       <iag:Metric id="int" type="integer"
     ↪ domain="[0..10000]"></iag:Metric>
15       <iag:Metric id="boolean" type="Boolean" domain="{true,
     ↪ false}"></iag:Metric>
16     </iag:Metrics>
```

```
17    </wsag:Context>
18    <wsag:Terms>
19      <wsag:All>
20        <wsag:ServiceDescriptionTerm wsag:Name="SDT_BirdAPI"
      ↪ wsag:ServiceName="BirdAPI"
21        iag:ServiceReference="
      ↪ http://papamoscas-isa.appspot.com/api/v3/birds">
22          <OfferItem name="PlanPrice" iag:Metric="price">9</OfferItem>
23        </wsag:ServiceDescriptionTerm>
24        <wsag:ServiceProperties wsag:Name="SP_BirdAPI"
      ↪ wsag:ServiceName="BirdAPI">
25          <wsag:VariableSet>
26            <wsag:Variable wsag:Name="Requests" iag:Metric="int">
27              <wsag:Location>/Requests</wsag:Location>
28            </wsag:Variable>
29          </wsag:VariableSet>
30        </wsag:ServiceProperties>
31        <wsag:GuaranteeTerm wsag:Obligated="Consumer"
      ↪ wsag:Name="RequestTerm">
32          <wsag:ServiceLevelObjective>
33            <wsag:CustomServiceLevel>Requests &lt;=
      ↪ 10</wsag:CustomServiceLevel>
34          </wsag:ServiceLevelObjective>
35        </wsag:GuaranteeTerm>
36      </wsag:All>
37    </wsag:Terms>
38    <wsag:CreationConstraints>
39    </wsag:CreationConstraints>
40  </wsag:Template>
```

Código C.15: Plantilla del acuerdo *basic plan* en WS-Agreement.

```
1   public class Helper {
2     private static final Logger LOG =
        ↪ Logger.getLogger(Helper.class.getName());
3     private static Helper instance;
4     private static AgreementManager agr;
5
6     protected Helper() {
7     }
```

```
8
9   public static Helper getInstance() {
10    if (instance == null) {
11      init();
12      instance = new Helper();
13      loadTestAgreements();
14    }
15    return instance;
16  }
17
18  public static AgreementManager getAgreementManager() {
19    return agr;
20  }
21
22  private static void init() {
23    agr = new AgreementManager();
24    agr.getStoreManager().registerFromFolder(new File("templates").
     ↪ getAbsolutePath(), false);
25  }
26
27  private static void loadTestAgreements() {
28    loadTestAgreement("basicPlanT", "basicUser1");
29    loadTestAgreement("basicPlanT", "basicUser2");
30    loadTestAgreement("mediumPlanT", "mediumUser1");
31    loadTestAgreement("mediumPlanT", "mediumUser2");
32    loadTestAgreement("proPlanT", "proUser1");
33    loadTestAgreement("proPlanT", "proUser2");
34  }
35
36  private static void loadTestAgreement(String templateName, String
     ↪ clientId) {
37    try {
38      agr.getStoreManager(). getAgreementTemplate(templateName).
     ↪ generateAgreementOffer(clientId). generateAgreement(clientId).
     ↪ register(clientId);
39    } catch (Exception e) {
40      LOG.log(Level.SEVERE, "Agreement Template " + templateName + "
     ↪ could not be loaded");
41    }
```

```
42    }
43
44    public Translator getIAgreeTranslator() {
45      return new Translator(new IAgreeBuilder());
46    }
47  }
```

Código C.16: Clase Helper con AML.

```
1   public class Helper {
2     private static final Logger LOG =
       ↪ Logger.getLogger(Helper.class.getName());
3     private static Helper instance;
4     private static AgreementManager agr;
5
6     protected Helper() {
7     }
8
9     public static Helper getInstance() {
10      if (instance == null) {
11        init();
12        instance = new Helper();
13        loadTestAgreements();
14      }
15      return instance;
16    }
17
18    public static AgreementManager getAgreementManager() {
19      return agr;
20    }
21
22    private static void init() {
23      agr = new AgreementManager();
24      agr.getStoreManager().registerFromFolder(new File("templates").
       ↪ getAbsolutePath(), false);
25    }
26
27    private static void loadTestAgreements() {
28      loadTestAgreement("basicPlanT", "basicUser1");
29      loadTestAgreement("basicPlanT", "basicUser2");
```

```
30      loadTestAgreement("mediumPlanT", "mediumUser1");
31      loadTestAgreement("mediumPlanT", "mediumUser2");
32      loadTestAgreement("proPlanT", "proUser1");
33      loadTestAgreement("proPlanT", "proUser2");
34    }
35
36    private static void loadTestAgreement(String templateName, String
       ↪ clientId) {
37      try {
38        agr.getStoreManager(). getAgreementTemplate(templateName).
       ↪ generateAgreementOffer(clientId). generateAgreement(clientId).
       ↪ register(clientId);
39      } catch (Exception e) {
40        LOG.log(Level.SEVERE, "Agreement Template " + templateName + "
       ↪ could not be loaded");
41      }
42    }
43
44    public Translator getIAgreeTranslator() {
45      return new Translator(new IAgreeBuilder());
46    }
47  }
```

Código C.17: Clase Helper

```
1   public class AgreementFilter implements Filter {
2
3     private static final Logger LOG =
       ↪ Logger.getLogger(AgreementFilter.class.getName());
4     private AgreementManager agr =
       ↪ Helper.getInstance().getAgreementManager();
5     private String clientId;
6
7     @Override
8     public void init(FilterConfig fConfig) throws ServletException {
9       ServletContext context = fConfig.getServletContext();
10    }
11
12    @Override
13    public void doFilter(ServletRequest request, ServletResponse
```

```
    ↪ response, FilterChain chain) throws IOException,
    ↪ ServletException {
14    HttpServletRequest req = (HttpServletRequest) request;
15    HttpServletResponse resp = (HttpServletResponse) response;
16    clientId = req.getParameter("user");
17    requestDone();
18    if (authorizeRequest()) {
19      chain.doFilter(request, response);
20      LOG.log(Level.INFO, "Request accepted");
21    } else {
22      resp.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
23      resp.sendRedirect("/error.html");
24      LOG.log(Level.INFO, "Request rejected");
25    }
26  }
27
28  @Override
29  public void destroy() {
30
31  }
32
33  private boolean authorizeRequest() {
34    Agreement agreement =
    ↪ agr.getStoreManager().getAgreement(clientId);
35    if (agreement != null) {
36      return agreement.evaluateGT("RequestTerm");
37    }
38    return false;
39  }
40
41  private void requestDone() {
42    Integer numReq;
43    Agreement agreement =
    ↪ agr.getStoreManager().getAgreement(clientId);
44    if (agreement != null) {
45      if (agreement.getProperty("Requests").getExpression() == null)
    ↪ {
46        agreement.setProperty("Requests", 0);
47      }
```

```
48        numReq = agreement.getProperty("Requests").intValue();
49        numReq++;
50        agreement.setProperty("Requests", numReq);
51      }
52    }
53  }
```

Código C.18: Clase AgreementFilter.

```
1  public class AgreementsServlet extends HttpServlet {
2
3    @Override
4    public void doGet(HttpServletRequest req, HttpServletResponse
      ↪ resp) throws IOException {
5      AgreementManager agr =
      ↪ Helper.getInstance().getAgreementManager();
6      Translator t = Helper.getInstance().getIAgreeTranslator();
7
8      String resourcePath = req.getPathInfo();
9      String agreement = "";
10
11     try {
12       if (resourcePath != null && !resourcePath.equals("/")) {
13         String[] resources = resourcePath.split("/");
14         if (resources.length > 2) {
15           resp.sendError(HttpServletResponse.SC_BAD_REQUEST, "Check
      ↪ URI");
16         } else {
17           String clientId = resources[1];
18           if (agr.getStoreManager().getAgreementMap().
      ↪ containsKey(clientId)) {
19             agreement = t.export(agr.getStoreManager().
      ↪ getAgreementMap().get(clientId));
20           } else {
21             resp.sendError(HttpServletResponse.SC_NO_CONTENT, "No
      ↪ data");
22           }
23         }
24       } else {
25         StringBuilder sb = new StringBuilder();
```

```
26          List<AgreementModel> models = new
    ↪ LinkedList<>(agr.getStoreManager().
    ↪ getAgreementMap().values());
27
28          Collections.sort(models, new Comparator<AgreementModel>() {
29            @Override
30            public int compare(AgreementModel o1, AgreementModel o2) {
31              return o1.getDocType().toString().
    ↪ compareToIgnoreCase(o2.getDocType().toString());
32            }
33          });
34
35          for (AgreementModel am : models) {
36            sb.append(t.export(am)).append("\n--------------\n");
37          }
38          agreement = sb.toString();
39        }
40        resp.setStatus(HttpServletResponse.SC_FOUND);
41        resp.setContentType("text/plain");
42        resp.setCharacterEncoding("UTF-8");
43        resp.getWriter().write(agreement);
44
45      } catch (Exception e) {
46        resp.setStatus(HttpServletResponse. SC_INTERNAL_SERVER_ERROR);
47        LOG.log(Level.WARNING, null, e);
48      }
49    }
50  }
```

Código C.19: Clase AgreementsServlet.

```
1  import es.us.isa.aml.*;
2
3  @ApplicationScoped
4  public class AMLHandler implements Serializable {
5
6    private static final long serialVersionUID = 1L;
7    private AgreementManager am;
8
9    @PostConstruct
```

```
10    public void init() {
11      am = new AgreementManager();
12    }
13
14    @Produces
15    public AgreementManager getAML() {
16      return am;
17    }
18
19    @Produces
20    public Store getStoreManager() {
21      return am.getStoreManager();
22    }
23
24    @Produces
25    public Translator getTranslator() {
26      return new Translator(new IAgreeBuilder());
27    }
28
29    @Produces
30    public Map<String, AgreementModel> getAgreementMap() {
31      return am.getStoreManager().getAgreementMap();
32    }
33
34    public void bindUserAndTemplante(String templateName, String
       ↪ token) {
35      am.getStoreManager(). getAgreementTemplate(templateName).
       ↪ generateAgreementOffer(token). generateAgreement(token).
       ↪ register(token);
36    }
37
38  }
```

Código C.20: Clase AgreementsServlet.

```
1  @Provider
2  @SLA
3  public class SLARequestFilter implements ContainerRequestFilter {
4    @Context
5    private ResourceInfo resourceInfo;
```

```
6   @Inject
7   private Store store;
8   @Inject
9   private Configuration config;
10
11  @Override
12  public void filter(ContainerRequestContext requestContext) throws
     ↪ IOException {
13   Boolean isSLA = false;
14   for (Annotation annotation :
     ↪ resourceInfo.getResourceMethod().getDeclaredAnnotations()) {
15     if (annotation instanceof SLA) {
16       isSLA = true;
17       break;
18     }
19   }
20   if (isSLA) {
21     //SLA aware method
22     List<String> token = requestContext.getUriInfo().
     ↪ getQueryParameters().get(
     ↪ config.getProperty("queryParamToken"));
23     String clientId;
24     if (token != null && !token.isEmpty()) {
25       clientId = token.get(0);
26       onRequestDone(clientId);
27       if (!isRequestAuthorized(clientId)) {
28         throw new WebApplicationException(
     ↪ Response.Status.PAYMENT_REQUIRED);
29       }
30     } else {
31       throw new
     ↪ WebApplicationException(Response.Status.BAD_REQUEST);
32     }
33   }
34  }
35
36  private boolean isRequestAuthorized(String clientId) {
37   Agreement agreement = store.getAgreement(clientId);
38   if (agreement != null) {
```

```
39        return agreement.evaluateGT( config.getProperty("requestGT"));
40      }
41      return false;
42    }
43
44    private void onRequestDone(String clientId) {
45      Integer numReq;
46      Agreement agreement = store.getAgreement(clientId);
47      if (agreement != null) {
48        if (agreement.getProperty(
     ↪ config.getProperty("requestVar")).getExpression() == null) {
49          agreement.setProperty( config.getProperty("requestVar"), 0);
50        }
51        numReq = agreement.getProperty(
     ↪ config.getProperty("requestVar")).intValue();
52        numReq++;
53        agreement.setProperty(c onfig.getProperty("requestVar"),
     ↪ numReq);
54      }
55    }
56
57  }
```

Código C.21: Clase SLARequestFilter.

```
1  @ApplicationPath("/")
2  public class ApplicationConfig extends Application {
3    @Override
4    public Set<Class<?>> getClasses() {
5      Set<Class<?>> resources = new java.util.HashSet<>();
6      addRestResourceClasses(resources);
7      return resources;
8    }
9    private void addRestResourceClasses(Set<Class<?>> resources) {
10     resources.add(es.us.isa.tfg.filters.SLARequestFilter.class);
11     resources.add(es.us.isa.tfg.services.AgreementsREST.class);
12   }
13 }
```

Código C.22: Clase ApplicationConfig.

```
1  @Stateless
2  @Path("agreements")
3  public class AgreementsREST {
4
5    @Inject
6    private AMLHandler amlHandler;
7
8    @Inject
9    private Translator translator;
10
11   public AgreementsREST() {
12   }
13
14   @GET
15   @Produces("text/plain")
16   public String findAll() {
17     StringBuilder sb = new StringBuilder();
18     List<AgreementModel> models = new
     ↪ LinkedList<>(amlHandler.getAML().getStoreManager().
     ↪ getAgreementMap().values());
19     Collections.sort(models, new Comparator<AgreementModel>() {
20       @Override
21       public int compare(AgreementModel o1, AgreementModel o2) {
22         return o1.getDocType().toString().
     ↪ compareToIgnoreCase(o2.getDocType().toString());
23       }
24     });
25     for (AgreementModel am : models) {
26       sb.append(getTranslator().export(am)). append("\n-----\n");
27     }
28     return sb.toString();
29   }
30
31   @GET
32   @Path("{clientId}")
33   @Produces("text/plain")
34   public Response find(@PathParam("clientId") String clientId) {
35     if (amlHandler.getAML().getStoreManager().
```

```
    ↪ getAgreementMap().containsKey(clientId)) {
36      return Response.ok(getTranslator().
    ↪ export(amlHandler.getAML().getStoreManager().
    ↪ getAgreementMap().get(clientId))). build();
37    } else {
38      return Response.status(Response.Status. NO_CONTENT).build();
39    }
40  }
41 }
```

Código C.23: Clase AgreementsREST.

```
1  @ApplicationScoped
2  public class APIConfiguration implements Serializable {
3    @PostConstruct
4    public void init() {
5      String in1 = getStringFromInputStream(Thread.currentThread().
     ↪ getContextClassLoader().
     ↪ getResourceAsStream("templates/basicPlanT"));
6      String in2 = getStringFromInputStream(Thread.currentThread().
     ↪ getContextClassLoader().
     ↪ getResourceAsStream("templates/mediumPlanT"));
7      String in3 = getStringFromInputStream(Thread.currentThread().
     ↪ getContextClassLoader().
     ↪ getResourceAsStream("templates/proPlanT"));
8
9      store.register(store.parseAgreementFile(in1));
10     store.register(store.parseAgreementFile(in2));
11     store.register(store.parseAgreementFile(in3));
12
13     loadTestAgreements();
14
15   }
16
17   private void loadTestAgreements() {
18     loadTestAgreement("basicPlanT", "basicUser1");
19     loadTestAgreement("basicPlanT", "basicUser2");
20     loadTestAgreement("mediumPlanT", "mediumUser1");
21     loadTestAgreement("mediumPlanT", "mediumUser2");
22     loadTestAgreement("proPlanT", "proUser1");
```

```
23      loadTestAgreement("proPlanT", "proUser2");
24    }
25
26    private void loadTestAgreement(String templateName, String
       ↪ clientId) {
27      try {
28        store.getAgreementTemplate(templateName).
       ↪ generateAgreementOffer(clientId).generateAgreement(clientId).
       ↪ register(clientId);
29      } catch (Exception e) {
30        e.printStackTrace();
31      }
32    }
33
34    public Map<String, AgreementModel> getAgreementModel() {
35      return store.getAgreementMap();
36    }
37  }
```

Código C.24: Clase APIConfiguration de *PapamoscasAPI*

BIBLIOGRAFÍA

[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. *Web Services Agreement Specification (WS-Agreement)*. 2007. (pages 102, 119).

[2] BOE. xvi convenio colectivo estatal de empresas de consultoría y estudios de mercados y de la opinión pública. *BOE*, 2009. (page 18).

[3] BOE. xiii convenio colectivo de ámbito estatal para los centros de educación universitaria e investigación. *BOE*, 2015. (page 18).

[4] A. M. Gutierrez, C. C. Marquezan, M. Resinas, A. Metzger, A. Ruiz-Cortes, K. Pohl, A. M. Gutierrez, C. C. Marquezan, M. Resinas, A. Metzger, and K. Pohl. Extending ws-agreement to support automated conformity check on transport and logistics service agreements. In *ICSOC*, volume 8274 of *Lecture Notes in Computer Science*, pages 567–574. Springer, 2013. doi: 10.1007/978-3-642-45005-1{_}47. (pages 5, 128).

[5] K. Kearney, F. Torelli, and C. Kotsokalis. Sla 2605;: An abstract syntax for service level agreements. In *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference ond*, October 2010. (page 102).

[6] K, Roland, K. Gregory, and W. Tinghe. A restful implementation of the ws-agreement specification. page pages, 2011. (page 86).

[7] M. Li, Q. Zhang, H. Chu, X. Hu, and F. Xu. Conha: An soa-based api gateway for consolidating heterogeneous ha clusters. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–3. IEEE, 9 2013. doi: 10.1109/CLUSTER.2013.6702645. (page 122).

[8] O. Martín. *Emparejamiento Automático de Servicios Web usando Programación con Restricciones*. phd, Dpto. de Lenguajes y Sistemas Informáticos, E.T.S. de Ingeniería Informática. Universidad de Sevilla, 2007. URL http://www.isa.us.es/sites/default/files/publication_44.pdf. (pages 5, 72, 129).

[9] C. Müller. *On the Automated Analysis of WS-Agreement Documents*. International dissertation, Universidad de Sevilla, June 2013. URL http://www.isa.us.es/sites/default/files/muller-Phd-PTB.pdf. (pages 72, 73, 75, 119).

[10] C. Muller, O. Martin-Diaz, M. Resinas, P. Fernandez, and A. Ruiz-Cortes. A ws-agreement extension for specifying temporal properties in slas. In *III Jornadas Científico-Técnicas en Servicios Web y SOA*, pages 77–84, 9 2007. (page 126).

[11] C. Muller, O. Martin-Diaz, A. Ruiz-Cortes, M. Resinas, and P. Fernandez. Improving Temporal-Awareness of WS-Agreement. In *5th. Intl. Conf. on Service Oriented Computing (ICSOC)*, volume 4749 of *LNCS*, pages 193–206, Vienna, Austria, Sept. 2007. Springer Verlag, Springer Verlag. ISBN 978-3-540-74973-8. doi: 10.1007/978-3-540-74974-5_16. URL http://www.springerlink.com/content/l36720u74nl4r2m5/. (page 126).

[12] C. Muller, A. Ruiz-Cortes, and P. Fernandez. Temporal-Awareness in SLAs. Why should we be concerned? In *Service-Oriented Computing - ICSOC 2007 Workshops: 1st. Non-Functional Properties and Service Level Agreements in Service Oriented Computing Workshop (NFPSLA)*, volume 4907 of *LNCS*, pages 166–173, Vienna, Austria, Sept. 2007. Springer Verlag, Springer Verlag. ISBN 978-3-540-93850-7. doi: 10.1007/978-3-540-93851-4_16. URL http://events.deri.at/nfpsla-soc07/. (page 126).

[13] C. Muller, A. M. Gutierrez, M. Resinas, P. Fernandez, A. Ruiz-Cortes, A. M. Gutierrez, and M. Resinas. iagree studio: A platform to edit and validate ws-agreement documents. In *11th Intl. Conf. on Service Oriented Computing (ICSOC)*, volume 8274 LNCS, pages 696–699, 2013. doi: 10.1007/978-3-642-45005-1{_}63. (pages 5, 102, 129).

[14] C. Muller, A. M. Gutierrez, O. Martin-Diaz, M. Resinas, P. Fernandez, and A. Ruiz-Cortes. Towards a formal specification of slas with compensations. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences*, volume 8841 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2014. doi: 10.1007/978-3-662-45563-0{_}17. (pages 5, 75, 126, 128).

[15] C. Muller, A. M. Gutierrez, M. Resinas, P. Fernandez, and A. Ruiz-Cortes. Towards compensable slas. In *4th International Workshop on Adaptive Services for the Future Internet (WAS4FI in ESOCC14)*, 2014. doi: 10.1007/978-3-319-14886-1{_}4. (pages 5, 126, 128).

[16] C. Muller, M. Oriol, X. Franch, J. Marco, M. Resinas, and A. Ruiz-Cortes. Comprehensive Explanation of SLA Violations at Runtime. *IEEE Transactions on Services Computing*, 7(2):168–183, 2014. ISSN 1939-1374. doi: 10.1109/TSC.2013.45. (page 86).

[17] C. Müller, M. Resinas, and A. Ruiz-Cortés. Automated analysis of conflicts in ws-agreement documents. *IEEE Transactions on Services Computing*, 7(4):530–544, October-December 2014. ISSN 1939-1374. doi: 10.1109/TSC.2013.9. (pages 5, 128).

[18] Q. Zhang, H. Chu, M. Li, and X. Hu. A unified api gateway for high availability clusters. In *Proceedings 2013 International Conference on Mechatronic Sciences, Electric Engineering and Computer (MEC)*, pages 2321–2325. IEEE, 12 2013. doi: 10.1109/MEC.2013.6885428. (page 122).